Titre: Title:	Implantation, comparaison et analyse des performances de l'estimateur fréquentiel Crozier sur différentes plates-formes
Auteur: Author:	Louis-Pierre Lafrance
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lafrance, L.-P. (2003). Implantation, comparaison et analyse des performances de l'estimateur fréquentiel Crozier sur différentes plates-formes [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7190/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7190/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

NOTE TO USERS

POLYTECHNIQUE DE MONTRÉAL

IMPLANTATION, COMPARAISON ET ANALYSE DES PERFORMANCES DE L'ESTIMATEUR FRÉQUENTIEL CROZIER SUR DIFFÉRENTES PLATES-FORMES

LOUIS-PIERRE LAFRANCE
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE EN SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
JANVIER 2003

Your file Votre référence
ISBN: 0-612-91952-8
Our file Notre référence
ISBN: 0-612-91952-8

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

IMPLANTATION, COMPARAISON ET ANALYSE DES PERFORMANCES DE L'ESTIMATEUR FRÉQUENTIEL CROZIER SUR DIFFÉRENTES PLATES-FORMES

Présenté par : LAFRANCE Louis-Pierre

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

A été dûment accepté par le jury d'examen constitué de :

M. AUDET Yves, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. BOYER, Ph.D., François, membre

REMERCIEMENTS

En premier lieu, je désire remercier mon directeur de recherche, Monsieur Yvon Savaria, pour le support financier dont j'ai bénéficié et surtout pour ses conseils avisés : nos discussions ont toujours été constructives et fructueuses. Son expérience a largement contribué à la réalisation de ce projet.

J'aimerais, également, remercier le Dr Pierre Lavoie, ainsi que son groupe de recherche au RDDC (recherche et Développement pour la Défense du Canada) pour leur collaboration soutenue. En plus d'agir à titre de parrain du projet, Monsieur Lavoie a directement collaboré à mes travaux. Son appui constant et ses conseils judicieux m'ont permis de mener à terme ce projet.

Un merci tout particulier s'adresse à Marc-André Cantin étudiant du GRM et collaborateur dans le projet parrainé par RDDC. Son support, son aide et ses conseils se sont avérés efficaces et essentiels.

Enfin, je ne voudrais pas oublier le personnel et les étudiants du GRM qui ont contribué à créer un environnement de travail harmonieux et plaisant.

RÉSUMÉ

Pour nombre d'applications de communication numérique en temps réel, telles que radars et sonars, l'estimation fréquentielle d'une tonalité dominante représente une fonction importante et souvent déterminante pour l'efficacité de traitement. La performance d'un estimateur fréquentiel dépend de sa capacité à produire rapidement une estimation précise à partir d'un court segment de signal, et ce, pour des fréquences allant jusqu'au taux de Nyquist.

L'algorithme de Crozier est un de ces estimateurs fréquentiels qui répondent particulièrement bien à ces critères de performance. En plus d'avoir une complexité proportionnellement linéaire aux nombres d'échantillons, ce qui facilite son implémentation tant logicielle que matérielle, Crozier se distingue des autres estimateurs par un seuil SNR (Signal Noise Ratio) inférieur et un rendement adéquat pour des fréquences proches de la fréquence de Nyquist.

L'importance d'un estimateur fréquentiel, tel que Crozier, au sein de nombreuses et diverses applications, fait de sa méthode de traitement un sujet de recherche crucial. L'implémentation de cet estimateur doit non seulement garantir une grande efficacité de traitement mais aussi, une certaine facilité d'intégration et de réutilisation. L'intégration de l'estimateur doit aussi tenir compte de supports allant des processeurs de hautes performances aux plates-formes de type système sur puce tout en considérant des formes d'implantations aussi diverses que matérielles, logicielles/matérielles et logicielles.

Deux approches ont été envisagées afin de réaliser une implémentation de l'estimateur Crozier et d'en analyser les performances. La première, de type logicielle, vise la puissance de calcul et l'accessibilité des processeurs commerciaux tels que Pentiums III et IV. Ces processeurs disposent d'extensions de jeux d'instructions (MMX, SSE1 et SSE2) qui offrent une efficacité de calcul intéressante. Diverses implémentations ont été produites afin de mesurer, sur de telles machines, les

performances de l'estimateur et d'évaluer les coûts associés à l'utilisation de jeux d'instructions optimisés.

La seconde, de type matériel, a pour objectif de créer un module réutilisable ou IP (Intellectual Property) qui pourra intégrer une plate-forme de type système sur puce reprogrammable. En plus d'une bonne puissance de calcul, le module créé offre une architecture cellulaire configurable qui facilite son intégration et sa réutilisation. Afin de valider l'implémentation, le module a été implanté sur la plate-forme de développement de système sur puce ARM Integrator.

Bien que l'implémentation matérielle offre la plus grande efficacité de calcul, les processeurs de la famille Pentium ont donné de surprenants résultats. Dans les deux cas, les performances sont excellentes. Les coûts initiaux relatifs à la conception et la réalisation sont, bien entendu, plus élevés pour l'implémentation matérielle. Toutefois, la nature réutilisable de cette implémentation ainsi que l'avènement des technologies de systèmes sur puces peut, dans certain cas, justifier cette approche.

ABSTRACT

Fast and precise frequency estimation of single-tone digitized signals are often crucial for applications in communication, radar, sonar and electronic warfare. The real-time performance of a single-tone frequency estimation algorithm depends not only on its computational efficiency, but also on its ability to yield accurate estimates from short signal segments at all frequencies up to Nyquist rate. A not-so-well-known algorithm developed by S. Crozier is quite powerful in this respect. It remains accurate at all frequencies, has a low SNR threshold, and its complexity is linearly proportional to the number of signal samples, which is well suited for both hardware and software implementations.

The motivations for an implementation of a frequency estimator such as Crozier are based on the importance of that functionality for many applications. Thus, the architectures build for the implementations must offer not only a good processing efficiency but should also be reusable to simplify the integration of the estimator in different applications.

A software implementation has, first, been considered. Modern processors such as Intel's Pentiums offer processing efficiency, accessibility and support that make them a very interesting solution for the Crozier estimator. For a few years now, these generic processors have offered extended optimized instructions, based on parallel calculation, that accelerate heavy processing. In order to access easily the optimized instructions, programmers can benefit from a variety of development facilities built by Intel. Several implementations of the Crozier estimator were developed. Their performances, on the Pentium processors, were measured. The efficiency of the programming environment was evaluated.

A hardware solution has also been investigated. A reusable hardware module or IP (Intellectual Property) that would fit any system on chip based application was constructed. The module involves, mostly, a completely configurable architecture that,

based on a cellular structure, allows a great deal of flexibility and variety. The hardware module was mounted on ARM Integrator system on chip development platform.

Although the hardware implementation offers the greatest processing efficiency, the Pentium processors offered quite surprising results. Both implementation methods are quite powerful in this respect. Initial costs involved with a hardware solution are higher than for software. On the other hand, the upcoming system on chip technologies and reusable nature of the module justifies the use of a hardware reusable module.

TABLES DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTES DES SIGLES ET ABBRÉVIATIONS

COORDIC		Coordinate Rotation Digital Computing
CRLB		Cramer-Rao-Lower-Bound
DMA		Delay-Multiply-Average
DMARAD		Delay-Multiply-Average- Rotate-Add-Decimate
dB	:	décibels
DSP	:	Digital Signal Processor / Processing
IA-32		32 bits Intel Architectures
IP	:	Intellectual Property
FFT	:	Fast Fourier Transform
FIFO	:	First In – First Out
FPGA	:	Field Programmable Gate Array
RAD	:	Rotate-Add-Decimate
LUT	:	Look Up Table
MDR	:	Methodology Design-Reuse
MKL	:	Math Kernel Library
MMX	:	MultiMedia eXtensions
RTDSC	:	Read Time Stamp Counter
SIMD	:	Single Instruction Multiple Data
SNR	:	Signal Noise Ratio
SPL	:	Signal Processing Library
SoC	:	System on Chip
SoP/RC	:	System on Programmable/Reprogrammable Chip
SSE	:	Streaming SIMD Extension

LISTES DES ANNEXES

CHAPITRE 1
INTRODUCTION

L'estimation fréquentielle à partir d'un ensemble d'échantillons complexes est une fonction essentielle à de nombreuses applications en communication numérique. Que ce soit pour des communications mobiles, satellite ou des applications militaires telles que la surveillance radar, la phase d'estimation fréquentielle, au cœur de la chaîne de traitement, a une incidence importante sur la performance du système.

La performance d'un estimateur fréquentiel dépend non seulement de son efficacité de calcul, mais aussi de sa capacité à produire de bonnes estimations à partir d'un court segment d'échantillons, et ce, pour une plage de fréquences assez grande. Plusieurs estimateurs ont été conçus afin de répondre adéquatement à ces critères de performance. Parmi ceux-ci, un estimateur du nom de Crozier est particulièrement efficace : en plus d'offrir une complexité qui croit linéairement avec le nombre d'échantillons, ses performances, au niveau de la précision de l'estimation, sont supérieures à celles de ses compétiteurs.

Le département de recherche en communication de RDDC (Recherche et Développement pour la Défense du Canada) utilise l'estimateur Crozier dans plusieurs de ses applications. Généralement, pour ces différentes applications, l'estimateur Crozier représente l'une des fonctions, sinon la fonction qui consomme la plus grande part du temps de calcul. Évidemment, la performance globale du système découle directement de la rapidité de traitement de ce dernier. Dans ce contexte, l'efficacité de calcul de l'estimateur Crozier est un paramètre dominant qui nécessite une stratégie d'implémentation particulière.

La problématique ne se situe pas seulement au niveau de l'efficacité de calcul, mais aussi au niveau de la diversité des applications dans lesquelles l'estimateur est susceptible d'intervenir. Plusieurs paramètres architecturaux sont déterminés selon

l'application et le contexte d'utilisation. Par exemple, le support matériel ne sera certainement pas le même dans une application destinée au cockpit d'un avion de chasse que dans un système de surveillance situé au sol, où il n'existe aucune restriction significative en matière d'espace et de dissipation de puissance. L'interface d'accès à l'estimateur est un autre exemple de paramètre susceptible de changer beaucoup selon l'application en cause. Les coûts associés à l'adaptation de l'estimateur en fonction des diverses applications sont considérables. Ainsi, en plus d'une excellente efficacité de calcul, l'implémentation de l'estimateur doit offrir une certaine portabilité, une facilité d'intégration et un caractère réutilisable.

Avec des circuits électroniques dont la capacité d'intégration ne cesse de croître, les solutions au traitement numérique d'algorithmes mathématiques sont de plus en plus nombreuses. Les processeurs commerciaux qui, jusqu'à maintenant, étaient restreints à des usages génériques offrent, aujourd'hui, des propriétés architecturales qui produisent une puissance de calcul surprenante. Leur grande popularité favorise leur accessibilité, la portabilité des applications et améliore le support accordé aux ingénieurs. Toutefois, leur grande consommation de puissance oblige, pour certaines applications, l'utilisation d'un autre support.

Parmi les solutions possibles, on doit nécessairement considérer les systèmes sur puces (SoC) et, particulièrement, les systèmes sur puces programmables ou reprogrammables (SoP/RC). Auparavant, les SoP/RC, généralement basés sur une technologie FPGA, étaient souvent utilisés à titre de plate-forme de développement. Aujourd'hui, les propriétés qu'ils offrent, en font des solutions particulièrement intéressantes pour beaucoup d'applications, notamment celles qui sont basées sur les chaînes de traitement numérique. En plus des performances physiques qui ont atteint un niveau plus qu'intéressant, ces plates-formes offrent des facilités telles que la disponibilité de bibliothèques de modules IP (*Intellectual Property*), de mémoire et de processeurs embarqués qui augmentent les performances et accélèrent la phase de conception.

1.1 EXEMPLE D'APPLICATION

L'algorithme IMOP (Intentionnal Modulation On Pulse) est un exemple d'application, parmi celles de RDDC, qui fait grand usage de l'estimateur Crozier. Cet algorithme s'inscrit dans un système de surveillance radar qui est composé typiquement de trois fonctions : l'échantillonnage des signaux interceptés, la caractérisation du signal échantillonné et l'identification de l'émetteur. Un diagramme bloc du système de surveillance est illustré à la Figure 1.1.

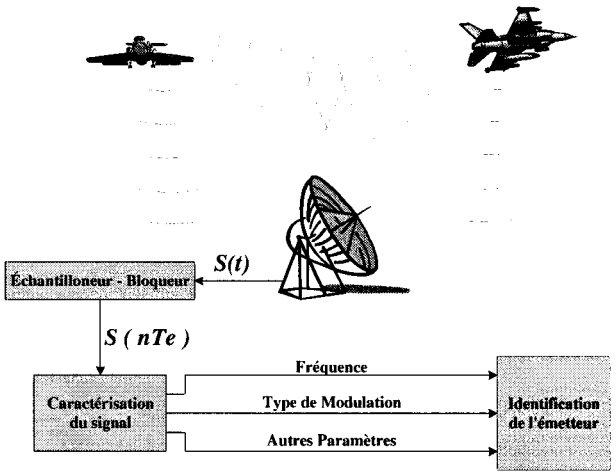

Figure 1.1 – Système de surveillance radar.

Les signaux radars émis par les différentes sources prennent la forme de courtes impulsions. Après leurs conversions en signaux numériques, ils sont dirigés vers le module de caractérisation. La caractérisation consiste à extraire certains paramètres et caractéristiques de signaux radars propres à un émetteur dont l'identité est souvent inconnue. Comme beaucoup d'applications en communication numérique, la phase de caractérisation est constituée d'une chaîne de traitement au sein de laquelle on peut trouver différentes étapes de filtrage, démodulation, détection d'impulsions, estimation fréquentielle, détection du type de démodulation et autres paramètres propres aux types de modulation détectée. La Figure 1.2 montre un schéma bloc de l'algorithme IMOP.

Finalement, à partir des paramètres extraits lors de la phase de caractérisation, le dernier module est en mesure de déterminer l'identité de l'émetteur.

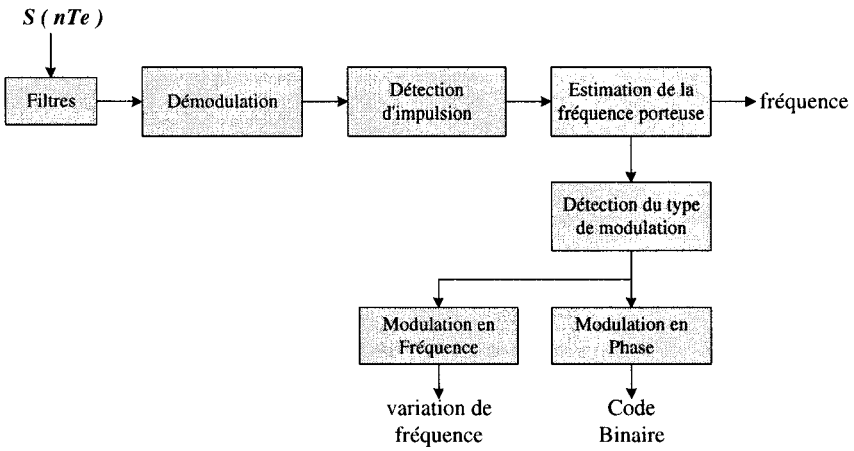

Figure 1.2 – IMOP.

L'algorithme de Crozier joue un rôle important dans l'application de IMOP. Des études de profilage ont démontré que, parmi toutes les fonctions de IMOP, Crozier était la fonction la plus fréquemment utilisée. Sur un processeur commercial typique, on estime que l'exécution de l'algorithme de Crozier consomme entre 75% et 80% du temps de calcul global de IMOP. Évidemment, IMOP est une application qui doit s'exécuter en temps réel. Il est donc essentiel que l'algorithme de Crozier ait une grande rapidité d'exécution.

1.2 OBJECTIFS

L'objectif de cette recherche est de réaliser et analyser diverses implémentations de l'algorithme de Crozier dont les spécifications répondent aux différentes applications auxquelles l'estimateur est susceptible de participer. Le critère de performance dominant est évidemment la vitesse à laquelle l'estimateur peut traiter des impulsions ou signaux. L'estimateur doit, également, offrir portabilité, accessibilité et facilité d'intégration et de réutilisation. Pour ce faire, deux approches ont été ciblées.

La première vise les principaux processeurs génériques, notamment ceux de la famille Pentium d'Intel que l'on retrouve en grand nombre sur le marché. Depuis quelques années, les principaux processeurs commerciaux ont des propriétés architecturales qui offrent une surprenante puissance de calcul. De plus, les compagnies comme Intel ont développé une gamme d'interfaces et de mécanismes de programmation qui permettent, via un langage de haut niveau comme le C, de mettre à profit cette puissance de traitement. Ainsi, en plus d'une efficacité de calcul, cette solution offre une certaine portabilité et ce, à des coûts d'implémentation relativement faible. Globalement, les objectifs sont, d'une part, d'évaluer les tenants et aboutissants relatifs aux méthodes de développement conçues pour ces processeurs et, d'autre part, de mesurer leurs performances. Éventuellement, les résultats de cette portion de la recherche serviront au développement d'une fonction Crozier, en vue de l'insérer dans une bibliothèque de modules de traitement de signal capable d'estimer la fréquence de porteuse d'un signal donné.

La seconde approche vise, quant à elle, une plate-forme de type système sur puce, où il est commun de retrouver un processeur de faible consommation de puissance associée à une portion de circuit dédié. La solution matérielle étant généralement celle qui permet d'atteindre les meilleures performances, la structure des systèmes sur puces se prête donc bien aux applications de type chaîne de traitement numérique. De plus, l'avènement des SoC et SoP/RC joint au concept de réutilisation et à l'existence de bibliothèques de modules IP, maintenant offertes aux concepteurs, changent les données quant aux coûts en temps et en argent associés à cette méthode de conception. En contrepartie, l'application du concept de réutilisation est assez complexe et nécessite une stratégie initiale bien définie. Principalement, l'objectif visé dans cette approche est d'élaborer une architecture matérielle, basée sur le parallélisme et le pipelinage qui, en plus d'offrir une très grande efficacité de calcul, intègre une bonne stratégie de réutilisation. En bout de ligne, cette recherche devrait fournir une architecture hautement générique, configurable et réutilisable.

1.3 ORGANISATION DU MÉMOIRE

L'algorithme décrivant l'estimateur de Crozier est présenté en détail au prochain chapitre. Nous profitons alors de l'occasion pour situer Crozier parmi les principaux estimateurs décrits dans la littérature. Suivra une analyse complète sur la complexité mathématique et le comportement de l'estimateur. L'objectif de cette analyse est de soulever ou de mettre en évidence les propriétés de l'algorithme qui auront un impact sur son implémentation. Nous, constaterons, entre autres, que certaines fonctions ont une importance capitale dans les performances de l'algorithme. Cette analyse conduit à des estimations d'efforts de calculs logicielles et matérielles. Celles-ci servent à évaluer le coût en effort de calcul et le gain potentiel que peuvent donner différentes implémentations.

Suite à l'analyse effectuée, nous traitons, au chapitre suivant, de reformulations, d'altérations ou de toutes modifications de l'algorithme qui permettront de simplifier la complexité et, ultimement, d'améliorer le rendement de l'algorithme de Crozier. Nous présentons, entre autres, une reformulation algorithmique qui permet une grande simplification de la complexité mathématique d'une portion de l'algorithme. Nous proposons, en outre, des alternatives qui, au prix de compromis, permettent de contourner certaines restrictions imposées par l'algorithme. À ce stade, tous les éléments sont en place pour procéder aux implémentations.

Les chapitres suivants proposent diverses implémentations de l'algorithme relatives aux approches ciblées. En premier lieu, nous démontrons l'approche logicielle basée sur les processeurs de la famille Pentium : divers mécanismes de programmation sont décrits et analysés, leurs performances respectives sont mesurées et analysées.

Par la suite, nous exposons l'architecture matérielle du IP Crozier construite selon les spécifications de performance et de réutilisation. Le module matériel réutilisable a été implanté sur une plate-forme, la plate-forme de développement ARM Integrator. Les performances ont aussi été mesurées et analysées.

CHAPITRE 2
L'ALGORITHME DE CROZIER

Il existe une panoplie d'estimateurs fréquentiels dont les propriétés, aussi variées que nombreuses, répondent aux différents besoins des multiples applications en traitement de signaux numériques. La performance d'un estimateur donné est généralement le résultat d'un compromis entre la précision de l'estimation et l'efficacité de calcul.

Une façon simple d'obtenir un bon indice de l'effort de calcul d'un estimateur consiste à évaluer la complexité mathématique de ce dernier. L'évaluation de la performance en précision est, quant à elle, basée sur une méthode statistique.

On calcule la variance des erreurs mesurées pour un ensemble d'estimations produites selon différents rapports signal sur bruit (SNR – *Signal noise ratio*). La courbe ainsi formée est comparée à la courbe de *Cramer-Rao* (*CRLB – Cramer-Rao-Lower-Bound*) [24] qui représente une limite inférieure théorique à la variance de l'erreur pour toute estimation fréquentielle dite non biaisée. Généralement, on distingue deux critères selon lesquels on évalue la performance en précision d'un estimateur : la différence en dB entre les courbes théoriques et mesurées et le seuil SNR de cette dernière. Le seuil SNR est le rapport du bruit pour lequel la courbe d'un estimateur s'éloigne de façon significative de la courbe théorique.

Pour les applications qui nous concernent, l'estimateur fréquentiel considéré doit nécessairement, avoir la capacité de converger rapidement vers un résultat. Il existe une série d'estimateurs dont la particularité est de pouvoir produire une estimation adéquate à partir d'un court segment du signal original. Ces estimateurs sont, pour la plupart, basés sur une méthode nommée *Pulse Pair*. Cette méthode ainsi que les principaux estimateurs reconnus dans la littérature sont décrits à la section suivante.

2.1 ESTIMATEURS FREQUENTIELS

Tretter [23] propose une solution classique à l'estimation fréquentielle d'un signal donné. La démarche consiste à minimiser, par la méthode des moindres carrés, l'erreur quadratique, due au bruit, sur la phase d'un signal. Pour de hauts rapports SNR, l'estimateur de Tretter rejoint en variance la courbe référentielle de *Cramer-Rao* (CRLB).

Considérons un signal quelconque S_n composé de N échantillons complexes dont le modèle mathématique est

$$S_n = Ae^{j(\omega_0 nT+\theta)} + r_n \tag{1}$$

L'amplitude A, la fréquence ω_0 et la phase θ sont des constantes réelles mais inconnues, r_n représente un bruit blanc inhérent au signal et T, la période d'échantillonnage. L'objectif est d'estimer la fréquence ω_0 du signal. Le fait que la phase et l'amplitude soient inconnues et le bruit sont des éléments nuisibles à l'estimation.

Il est possible, sous certaines conditions [23][23], de remplacer le modèle du signal, décrit ci-dessus, par l'approximation suivante;

$$S_n \cong Ae^{j(\omega_0 nT+\theta+v_n)} \tag{2}$$

où v_n représente une composante de phase équivalente au bruit blanc r_n. La phase ϕ_n du signal est définie par

$$\phi_n = \omega_0 nT + \theta + v_n \tag{3}$$

Tretter suggère alors d'utiliser la méthode des moindres carrés (*least squares*) afin d'évaluer les valeurs des constantes ω_0 et θ qui minimisent l'erreur quadratique due à la composante de bruit;

$$\hat{\omega}_0, \hat{\theta} \leftarrow \min_{\omega,\theta} \sum_{n=0}^{N-1} \left\{\phi_n - \omega_0 T\left[n-(N-1)/2\right]-\theta\right\}^2 \tag{4}$$

Le problème majeur relié à cet estimateur provient de la difficulté à déterminer la fonction de phase ϕ_n. Pour résoudre celle-ci, un algorithme de déroulement de phase (*phase unwrapping*) doit être appliqué sur l'ensemble des phases des échantillons. Cette manipulation a pour effet d'accroître considérablement la complexité de l'estimateur. De plus, les algorithmes de déroulement de phase sont relativement peu efficaces pour des rapports SNR faibles.

On peut éviter le problème de déroulement de phase en travaillant sur la différence de phases entre les échantillons successifs plutôt qu'avec les phases elles-mêmes. Le déphasage observé entre deux échantillons successifs du signal S_n donne la proportion de rotation du signal au cours d'une période d'échantillonnage. Ceci se traduit par

$$\hat{\omega}T = \phi_{n+1} - \phi_n \tag{5}$$

où $\phi_n = \angle S_n$.

Une autre façon de calculer le déphasage consiste à utiliser le produit conjugué entre les échantillons successifs : $\angle\left(S_n^* S_{n-1}\right)$. Si on prend la moyenne des différences de phases pour l'ensemble du signal, on obtient une estimation de la fréquence porteuse du signal, soit

$$\hat{\omega}T = \frac{1}{N-1}\sum_{n=1}^{N-1} \angle\left(S_n^* S_{n-1}\right) \tag{6}$$

Cet estimateur porte le nom de *Pulse-Pair* [1] et son principal avantage est d'éliminer le phénomène de déroulement de phase du processus d'estimation fréquentielle. On retrouve dans la littérature une série d'estimateurs qui sont tous des variantes plus ou moins améliorées de cette méthode; ils font tous abstraction du problème de déroulement de phase et ils ont tous une performance en précision au moins équivalente à celle de l'estimateur de Tretter (*least-squares*).

Parmi ceux-ci, on retient l'estimateur de Kay [15] qui décrit l'estimation fréquentielle non pas une comme une moyenne des déphasages successifs, mais comme

une *moyenne pondérée* des déphasages successifs. L'estimateur de Kay, comme les deux autres, rejoint la courbe CRLB à des rapports SNR élevés. Il offre, toutefois, un seuil SNR beaucoup plus bas [1][15] que ses compétiteurs. L'estimateur de Kay est

$$\hat{\omega}T = \sum_{n=1}^{N-1} w_n \left(\phi_n - \phi_{n-1}\right) \tag{7}$$

où w_n est le nième poids de pondération. La fonction de poids pondérés, telle que définie par Kay [15], est

$$w_n = \frac{3/2N}{N^2-1}\left\{1-\left[\frac{n-\left(\frac{N}{2}-1\right)}{\frac{N}{2}}\right]^2\right\} \tag{8}$$

L'algorithme de Crozier est également une variante intéressante de la méthode *Pulse-Pair.* Sa particularité est qu'il travaille de bout en bout sur les échantillons complexes plutôt que sur les phases ou différences de phases. La nuance est mince mais donne d'avantageux résultats de performance. Le calcul principal de l'estimateur se résume ainsi

$$e^{jd\hat{\omega}T} \sim \sum_{n=0}^{N-1-d} S_n^* S_{n+d} \tag{9}$$

où d représente un délai exprimé en période d'échantillonnage. L'estimateur de Crozier sera présenté en détail à la prochaine section.

Les estimateurs présentés ont tous une complexité O(N). Le principal avantage de l'algorithme de Crozier est qu'il est efficace sur une plage de fréquences et de rapports SNR beaucoup plus large que les autres. La Figure 2.1 donne les résultats de

performance obtenus par les estimateurs de Tretter, Kay et Crozier[1]. Les courbes montrent la variance des erreurs mesurées, pour un ensemble d'estimations, selon différents rapports signal sur bruit (SNR – *Signal noise ratio*). La performance est évaluée en comparant chaque courbe à la courbe théorique de *Cramer-Rao*. Les mesures ont été prises pour un signal de 40 échantillons et des fréquences de $fT = 0.05$ et $fT = 0.49$ où fT est le produit de la fréquence de la porteuse du signal et de la période d'échantillonnage.

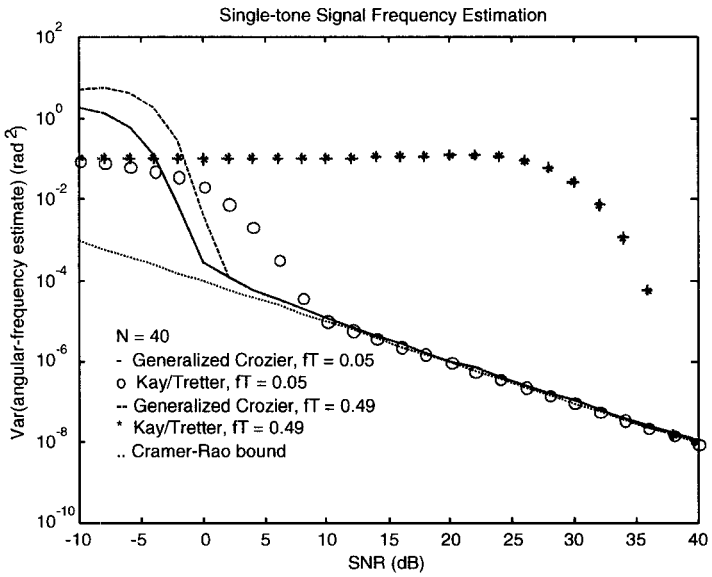

Figure 2.1 – Variance de l'erreur pour différents estimateurs et conditions d'opération.

On constate, sur la figure, que pour des rapports SNR élevés et une fréquence relativement éloignée du taux de Nyquist ($fT_{Nyquist}$=0.5), les trois estimateurs rejoignent, en variance, la courbe de *Cramer-Rao* ; la différence est de 0.5 dB [4][19], ce qui est

[1] Les résultats montrés sur cette figure ont été produits et fournis par l'équipe du Dr Pierre Lavoie à RDDC (Recherche & Développement Défense Canada).

pratiquement négligeable. L'estimateur de Crozier atteint, toutefois, la courbe à un rapport SNR inférieur (2 à 3 dB) à celui des deux autres. Cette différence est encore plus significative pour des fréquences s'approchant du taux de Nyquist. Par exemple, pour une fréquence de fT=0.49, le seuil SNR de Crozier est d'environ une trentaine de dB inférieur à ceux des estimateurs de Kay et de Tretter. L'estimateur de Crozier peut donc fournir une estimation de qualité à une fréquence très proche du taux de Nyquist et un rapport signal sur bruit assez faible.

Notons que ces estimateurs ont également été comparés à la traditionnelle transformée de Fourier rapide (FFT-*Fast Fourier Transform*) [4][19][21]. Les résultats ont démontré qu'en termes de précision, la FFT pouvait rivaliser avec les estimateurs de Kay et Tretter. En contrepartie, la FFT nécessite un effort de calcul beaucoup plus grand [4][19], ce qui rend cette solution très peu attrayante.

2.2 ALGORITHME DE CROZIER

2.2.1 Description générale

L'algorithme de Crozier fut introduit par Stewart N. Crozier en 1992 [4][19] Naturellement, la fonction de l'algorithme est d'estimer la fréquence porteuse d'un signal numérique donné. L'estimation proprement dite est produite à l'aide d'une technique appelée *delay-multiply-average* ou DMA qui découle directement de la méthode *Pulse Pair*.

Considérons une séquence de N échantillons complexes S_0, S_1,..., S_n,..., S_{N-1} et leurs phases respectives ϕ_0, ϕ_1, ..., ϕ_n,..., ϕ_{N-1}. Le principe de la fonction DMA, dont le comportement angulaire est schématisée à la Figure 2.2, consiste à estimer la moyenne des différences de phases entre des échantillons distancés d'un certain délai d, quantifié en période d'échantillonnage. Le déphasage est obtenu par le produit conjugué des

échantillons S_n et S_{n+d}. Le résultat donne le phaseur[2] complexe Z dont la phase est égale à $d\hat{\omega}T$. En extrayant une racine d'ordre d à ce dernier, on obtient le phaseur $\hat{W}$ qui représente la moyenne des différences de phases entre les échantillons *successifs* du signal. Le délai et la période d'échantillonnage étant des éléments connus de l'équation, il est donc possible de déduire la fréquence moyenne $\hat{f}_{signal}$ du signal. Les trois équations suivantes résument bien le comportement de la fonction DMA.

$$Z = \sum_{n=0}^{N-1} S_n^* S_{n+d} \sim e^{jd\hat{\omega}T} \tag{10}$$

$$\hat{W} = Z^{1/d} \tag{11}$$

$$d\hat{\omega}T = d * 2\pi * \frac{\hat{f}_{signal}}{f_{échantillonnage}} \tag{12}$$

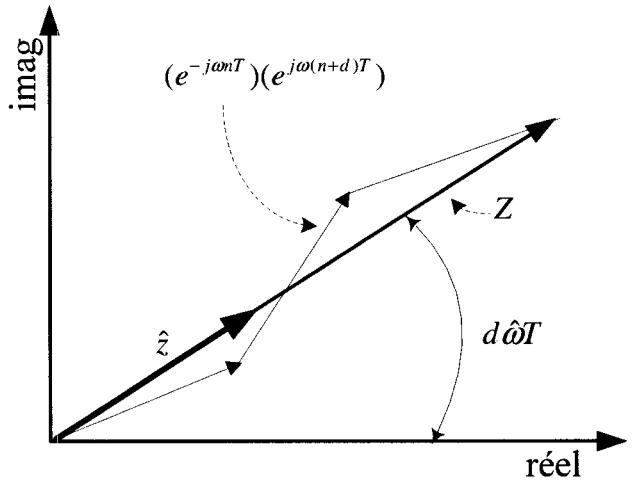

Figure 2.2 – Fonction *DMA* sous forme vectorielle

[2] Un vecteur complexe indiquant une variation de phase sur le cercle trigonométrique.

Une étude approfondie, effectuée par Lank, Reed et Pollon [17], a démontré que pour des valeurs de délais de *2N/3* et de *N/3*, la variance de l'erreur enregistrée pour l'estimation est au minimum, la précision de l'estimation est alors optimale. Cependant, l'application simple et directe de la fonction DMA pour ces délais peut conduire à des cas d'enroulement de phase. En effet, si on considère des fréquences d'échantillonnages supérieures à Nyquist, la phase estimée pour un délai plus grand que 1 peut être décrite par $d\hat{\omega}T + 2k\pi$ où k est un facteur d'enroulement de phase. Dans ce cas, seul un algorithme de déroulement de phase peut déterminer la bonne valeur de k. Pour prévenir le plus possible l'enroulement de phase, Crozier [4][19] utilise une forme itérative ou plusieurs estimations sont produites à partir de différents délais, dont le délai initial est de un et les délais suivants doublent à chaque itération. Le nombre d'itérations doit être suffisant pour que les valeurs de délais optimales soient utilisées. Il est évident que si Nyquist est respecté, le déphasage maximum pour un délai de 1 est inférieur à π [16]. C'est en s'appuyant sur une estimation initiale valide que Crozier a construit un algorithme qui, à partir de celle-ci, conduit les autres estimations vers les délais optimaux.

Normalement, puisque le délai double à chaque itération, l'estimation de la phase devrait également doubler. On peut donc facilement calculer, sans avoir à se soucier de l'enroulement, la différence entre la phase estimée au cours d'une itération et le double de celle obtenue à l'itération précédente. Crozier utilise, de façon pondérée, les différences d'estimations successives afin d'effectuer, d'itération en itération, un ajustement sur la phase initiale. Le tout, donne une sorte de moyenne pondérée de l'ensemble des estimations. Si les différences mesurées sont relativement petites, tout problème d'enroulement de phase sera évité. Généralement les grandeurs de différences dépendent du rapport signal sur bruit. La Figure 2.3 illustre un exemple d'ajustement appliqué au phaseur $\hat{W}_b$. On constate que le nombre de tours (enroulement) qu'aurait pu faire $\hat{z}_{b+1}$ ne change rien à la différence entre ce dernier et $\hat{z}_b^2$. Seule une grandeur excessive de l'erreur, ε_b, pourrait perturber l'estimation.

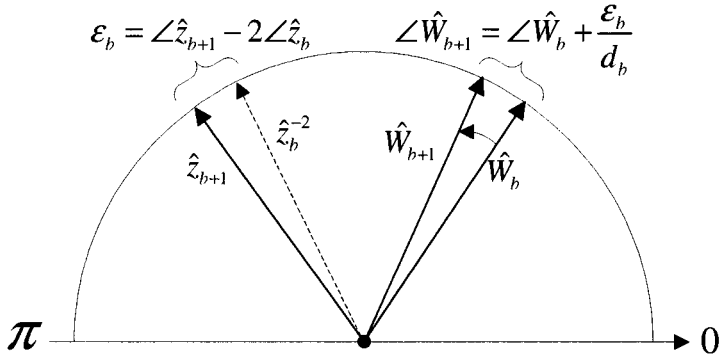

Figure 2.3 – Calcul de moyenne pondérée.

L'augmentation du délai par un facteur de deux se traduit concrètement par une décimation du signal d'un facteur de deux. La décimation est produite par le biais de la technique *Rotate-Add-Decimate* ou RAD [4][19]. En plus d'une décimation, la fonction RAD applique une correction vectorielle aux échantillons. La fonction consiste à générer un échantillon à partir de deux échantillons successifs du signal, dont l'un des deux a préalablement subi une rotation vectorielle d'un angle égal à la phase estimée.

Considérons la séquence d'échantillons $S_{2n-1,b}$ et $S_{2n,b}$ d'un signal donné à la branche b de l'algorithme, ainsi que leurs phases $\phi_{2n-1,b}$ et $\phi_{2n,b}$. Comme le montre le schéma de la Figure 2.4, la correction vectorielle consiste à faire pivoter le second échantillon de la séquence d'un angle égal à la phase $\angle\hat{z}_b$. La rotation est obtenue par le produit conjugué $\hat{z}^* S_n$. La longueur du vecteur corrigé dépend des composantes vectorielles des vecteurs $\hat{z}^*$ et S_n.

Puisque la phase $\angle\hat{z}_b$ correspond au déphasage moyen des échantillons du signal, la rotation appliquée à l'échantillon S_{2n} a pour effet de ramener ce dernier à un angle semblable à ceux de $\hat{z}_b$ et S_{2n-1}. L'addition du vecteur corrigé à celui du premier échantillon de la séquence devrait donner, tel qu'illustré à la Figure 2.5, une résultante se rapprochant de la phase estimée. Ainsi, le nouvel échantillon $S_{n,b+1}$ du signal de

l'itération *b+1* a été aligné selon l'angle du phaseur $\hat{z}_b$ estimé à la branche précédente. Cette opération a pour effet de doubler le rapport SNR, tout en réduisant de moitié le nombre d'échantillons à traiter [16]. Les effets de cette fonction, tant sur la précision de l'estimation que sur l'effort de calcul, seront exposés un peu plus loin au cours de ce chapitre. Le lecteur pourra constater que, malgré le coût élevé en terme d'effort de calcul, la technique RAD est absolument essentielle à l'algorithme de Crozier.

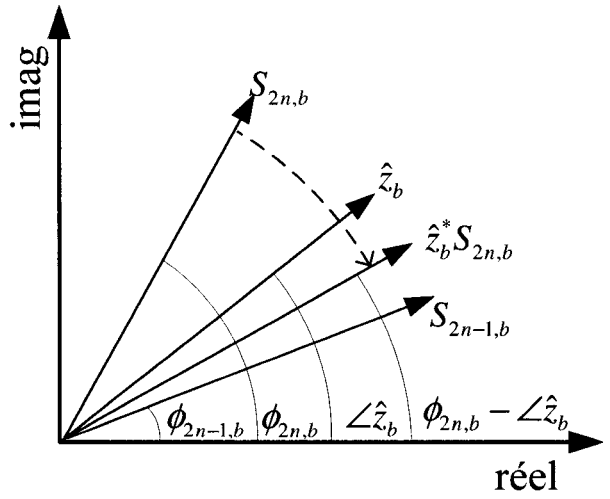

Figure 2.4 – Correction vectorielle de la Fonction RAD.

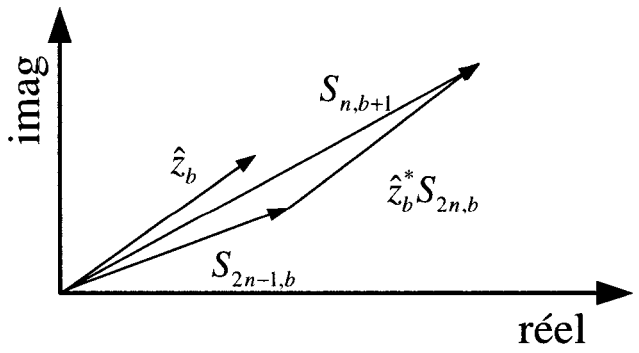

Figure 2.5 – Formation d'un nouvel échantillon.

2.2.2 Description détaillée

Le diagramme bloc de l'algorithme de Crozier est illustré à la Figure 2.6. Le nombre d'échantillons du signal est restreint à trois fois une puissance de deux, soit $N = 3x2^B$. Cette condition nous assure que le délai, qui double à chaque itération, atteindra bien les valeurs recherchées de $2N/3$ et $N/3$. La nombre nécessaire d'itérations, ou branches, telles que Crozier les nomme, est de $B+2$ où $B=log_2(N/3)$.

Comme nous l'avons mentionné précédemment, le délai, qui est initialement de 1, double de branche en branche. Ainsi, pour la branche b des $B+2$ branches totales à exécuter, le délai est égal à

$$d_b = 2^{b-1},\ b = 1,2,\ldots,B+2. \tag{13}$$

Puisque le signal est décimé par un facteur de deux à chaque itération, le nombre d'échantillons de ce dernier, pour une branche b donnée, est de

$$N_b = N / d_b,\ b = 1,2,\ldots,B+1. \tag{14}$$

À chaque branche l'estimation fréquentielle du signal est effectuée. Puisque, pour chaque itération, le délai double alors que le signal est décimé d'un facteur de deux, le produit conjugué de la fonction DMA est, généralement, appliqué aux échantillons successifs de la séquence. L'équation mathématique de la fonction est donc

$$Z_b = \begin{cases} \sum_{n=0}^{N_b-1} S^*_{n,b} S_{n+1,b},\ b = 1,2,\ldots,B+1, \\ S^*_{1,B+1} S_{3,B+1},\ b = B+2, \end{cases} \tag{15}$$

Il est à noter que pour la dernière itération, ce sont les premier et troisième échantillons de la séquence qui sont multipliés dans le produit conjugué. La relation mathématique de la décimation produite par la fonction RAD est

$$S_{n,b+1} = S_{2n,b} + \hat{z}^*_b S_{2n+1,b} \tag{16}$$

où $b=1,2,\ldots,B$, $n=1,2,\ldots,N_{b+1}$ et $\hat{z}$ est le vecteur normalisé de l'estimation. La normalisation est simplement obtenue par

$$\hat{z}_b = Z_b / |Z_b|, \; b = 1,2,\ldots,B+2. \tag{17}$$

La fonction RAD est utilisée jusqu'à la branche B uniquement. Pour les deux dernières itérations, le nombre d'itérations n'est plus que de trois et la décimation n'est plus nécessaire. Puisque le nombre d'échantillons est réduit de deux à chaque itération, la complexité générale de l'algorithme de Crozier est de O(N).

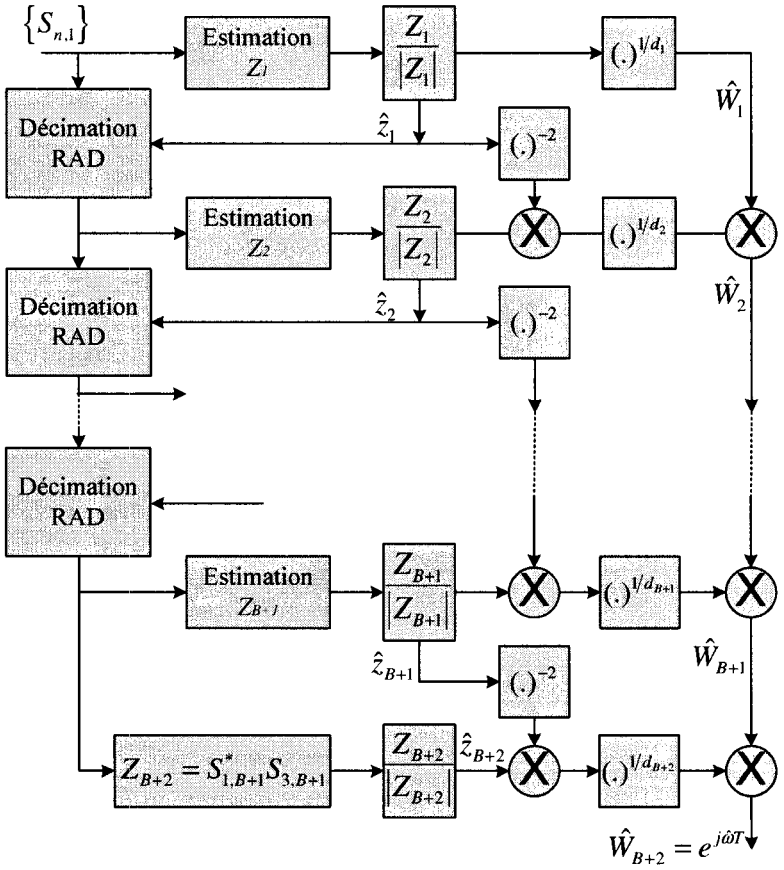

Figure 2.6 – Diagramme bloc de l'algorithme de Crozier

La méthode itérative dérivée par Crozier permet d'éviter, dans beaucoup de cas, l'enroulement de phase. Afin d'illustrer plus clairement le fonctionnement de la méthode, nous allons examiner le processus de calcul appliqué aux phaseurs normalisés $\hat{z}_1$ et $\hat{z}_2$ des deux premières branches. Mais avant tout, il faut rappeler que, puisque le délai double aux changements de branches, nous devons nous attendre à ce que l'estimation soit, également, approximativement doublée. La phase $\angle\hat{W}_1$, tributaire de la fréquence porteuse du signal, est obtenue en extrayant la racine d_1 du phaseur $\hat{z}_1$. Lors du passage de la première à la deuxième branche, $\hat{z}_2$ est divisé par le carré de $\hat{z}_1$. L'angle du quotient représente ainsi la différence entre le double de $\hat{z}_1$ et $\hat{z}_2$. En extrayant la racine d'ordre $1/d_2$, avec d_2=2, on obtient la moitié de cette différence, qui est ensuite multipliée lors de l'estimation initiale donnée par $\hat{W}_1$. Le résultat donne le phaseur ajusté $\hat{W}_2$. Le même processus se répète à chaque itération. On multiplie ainsi, à l'estimation initiale, la moitié, le quart, le huitième des différences entre les estimations successives. On applique, en quelque sorte, une série de corrections sur l'estimation initiale, ce que l'on peut généraliser par

$$\hat{W}_b = \left(\hat{z}_b \cdot \hat{z}_{b-1}^{-2}\right)^{1/d_b} * \hat{W}_{b-1} \tag{18}$$

Comme nous l'avons mentionné un peu plus haut, si la différence entre les estimations successives est relativement petite, tout problème relatif à l'enroulement de phase sera évité. De plus, afin que ce calcul demeure juste, il est nécessaire de présumer que la différence enregistrée soit comprise entre $-\pi$ et π. Dans le cas contraire, il est impossible de déterminer le sens de la différence. Prenons, par exemple, deux angles de 30 et 40 degrés obtenus aux itérations b et $b+1$. Si aucun postulat n'est posé, la différence peut aussi bien être de 10 que de 350 degrés selon le sens donné. Aucune indication ne peut nous permettre de distinguer les des deux cas. On espère, ou on exploite une connaissance *a priori* du problème, pour faire en sorte que les erreurs sur les estimations soient inférieures à $-\pi$ et π.

2.3 ANALYSE ET ESTIMATION DE L'EFFORT DE CALCUL

Cette section vise à mettre en relief toute propriété, tout caractère ou comportement particulier de l'algorithme susceptible d'influencer les architectures et les performances relatives aux implémentations. Deux facteurs sont généralement déterminant pour la performance résultante d'une implémentation : la complexité mathématique et les dépendances de données. La complexité est généralement très représentative de l'effort de calcul nécessaire au traitement d'un algorithme. Elle tient compte autant du nombre d'opérations que de leurs complexités mathématiques. Les dépendances de données, quant à elles, relèvent directement du comportement de l'algorithme. Elles représentent souvent un facteur limitant au degré de traitement en parallèle. L'analyse des dépendances de données donne généralement un bon indice de l'optimisation possible.

Ces analyses nous permettront, d'une part, d'évaluer l'effort de calcul attribuable aux implémentations logicielles et matérielles et, d'autre part, d'établir les reformulations algorithmiques nécessaires. Mais avant tout, il est nécessaire, à ce stade, de définir certaines considérations préliminaires qui seront utiles tout au long de ce mémoire.

2.3.1 Considérations préliminaires

2.3.1.1 Références logicielles

Le présent projet traite en grande partie de l'implémentation d'un algorithme mathématique. Rappelons que l'objectif est de formuler différentes implémentations et d'en évaluer les tenants et aboutissants, spécialement au niveau de l'effort de calcul. En conséquence, tout au long de ce mémoire, il sera souvent question de temps de calcul, de délai, de complexité mathématique, de complexité de traitement, de nombre de cycles d'horloge ainsi que de bien d'autres termes relatifs au traitement numérique. Il s'avère donc très pratique de disposer, dès le départ, de certaines références en matière de traitement de données logicielles. Celles-ci permettront, entre autres, de produire des estimations et des analyses préliminaires.

De plus, la technologie moderne évolue si rapidement qu'il est difficile de savoir en tout temps les performances que peuvent offrir les différentes plates-formes de traitement. Cette analyse vise également à évaluer le potentiel d'un processeur générique, standard et qui est facilement disponible sur le marché. La première phase du projet consiste donc à cibler une plate-forme de référence et à profiler les performances pour différentes opérations de base telles que les addition, multiplication, exponentiation, etc.

Pour leurs popularité, ressources et support technique, les processeurs de la famille *Pentium* d'*Intel* se révèlent un excellent choix. Cette famille de processeurs dispose d'un compteur *Time Stamp* auquel on peut accéder par l'instruction **RDTSC** (*Read Time Stamp Counter*). Ce compteur enregistre de façon précise le nombre de cycles d'horloge écoulé. Ce compteur est constitué d'un registre de 64 bits (MSR- *model specific register*) qui est incrémenté à chaque cycle d'horloge. Une commande de remise à zéro permet de réinitialiser le registre. Le tout est géré par de petites routines assembleurs. Un exemple de code est fourni en ANNEXE A. Généralement, deux routines, *starchrono()* et *stopchrono(),* servent à déclencher ou stopper le chronomètre. Ces routines ne font qu'enregistrer, à un moment précis, le contenu du registre MSR dans une variable de 64 bits (*_int64*). Entre les deux routines, on insère la fonction à mesurer comme le montre les échantillons de code des Figure 2.7 et Figure 2.8.

La mesure est toutefois très sensible. Plusieurs facteurs tels que le compilateur, la mémoire cache, les opérandes utilisées et le type d'assignation peuvent affecter le résultat de la mesure. L'addition, par exemple, nécessite de 4 à 54 (et plus) cycles d'horloge selon l'utilisation que l'on en fait. Pour l'addition de deux vecteurs de 1000 valeurs, le délai est de 12 cycles par opération alors que, pour des vecteurs de 100000 valeurs, le délai observé est de 54 cycles par opération. Nous pouvons présumer que cette différence est due à un débordement de la cache.

De façon empirique, deux méthodes de mesure ont été mises sur pied afin de donner une évaluation juste des délais par opération. La première, dont le code se trouve à la

figure suivante, mesure le délai d'une opération appliquée à des variables d'un certain type. L'objectif, ici, est d'obtenir une mesure brute du délai de l'opération.

```
   Int64 c1=0; int KK=10000;
...
   for (j=0; j<KK; j++)
   {
      startChrono(c1);
         a=b OP c;
      stopChrono(c1);
   }
   m1 = ((j-1)/j)*m1 + (1/j)*c1;
...
```

Figure 2.7 – Profilage de l'opération OP.

La seconde, dont le code est décrit à la Figure 2.8, mesure le délai d'une opération appliquée aux données des vecteurs d'une certaine dimension. Puisque dans la majorité des cas, ce sont des vecteurs qui sont manipulés, il est intéressant de connaître le délai des opérations sous ces conditions.

```
   Int64 c1=0;
   int K= 1000, KK=10000;
...
   for (j=0; j< KK; j++)
   {
      startChrono(c0);
         for(i=0;i<K;i++)
            {}
      stopChrono(c0);

      startChrono(c1);
         for(i=0;i<K;i++)
            a[i]=b[i] OP c[i];
      stopChrono(c1);

      m0 = ((j-1)/j)*m0 + (1/j)*c0;
      m1 = ((j-1)/j)*m1 + (1/j)*c1;
   }
   resultat = (m1-m0)/K;
```

Figure 2.8 – Profilage en mode vecteur.

Les mesures ont toutes été effectuées pour des types de données réels (virgule flottante). Lorsque l'opération le permet, le comportement en nombres entiers (virgule fixe) est aussi mesuré. La mesure est effectuée plusieurs fois. La moyenne des mesures donne le nombre de cycles d'horloge par opération.

Le Tableau 2.1 présente les mesures de délai en cycles d'horloge pour un ensemble d'opérations choisies spécialement en fonction des besoins de l'algorithme de Crozier : addition (+), multiplication (*), exponentiation (exp()), racine carrée (sqrt()), décalages (>>) et arctangente (atan2()). Les trois dernières opérations sont générées à l'aide de fonctions appartenant à la bibliothèque mathématique (*math.h*) du langage C. Le processeur est un Pentium III de 866 MHz avec un bus de système à 133 MHz, une RAM de 128MB et une cache de 256Ko. Des vecteurs de 1000 échantillons sont utilisés. À moins d'avis contraire, ces conditions de mesures seront respectées tout au long du mémoire.

Tableau 2.1 – Mesures de profilage pour le *Pentium III*

	+	*	/	>>	Exp()	Sqrt()	Atan2()
Virgule flottante							
vecteur	8	8	26	---	340	85	310
variable	4	6	33	---	324	80	330
Virgule fixe							
vecteur	7	6	31	7	----	----	----
variable	3	5	39	4	----	----	----

Note : Ces résultats contiennent les temps requis pour les opérations et les assignations aux variables.

Nous pouvons observer une différence importante entre les deux formes de mesure pour les opérations d'addition et de multiplication. Sous un mode vecteur, ces opérations nécessitent presque le double du nombre de coups d'horloge. Ainsi, les deux méthodes de profilage nous donnent, en quelque sorte, les bornes inférieures et supérieures du

délai nécessaire au traitement des opérations. Notons également que le délai moyen enregistré pour la boucle « *for* » est de 5 cycles. Pour l'exécution d'un vecteur de dimension K, il faut donc compter

$$D_{\overline{OP}} = K(D_{OP} + D_{for}) \tag{19}$$

où D_{OP} et D_{for} sont, respectivement les délais accordés aux traitements de l'opération OP et de la boucle « *for* ». Finalement, soulignons que les calculs en virgule fixe des opérations d'addition et de multiplication sont légèrement plus rapide que ceux en nombres réels.

2.3.1.2 Précision et méthode d'évaluation

Le paramètre fondamental d'un estimateur fréquentiel est la précision avec laquelle il peut produire des estimations. Que l'efficacité d'un estimateur soit considérée en rapport avec sa justesse ou la rapidité avec laquelle il peut produire les estimations, la précision fait toujours partie de l'équation. Bien que traiter de la performance en termes de précision ne soit pas l'objet de ce mémoire, il est important de réaliser que toute reformulation, modification ou approximation induite par le traitement numérique de l'estimateur aura nécessairement un impact sur la précision. Il est donc primordial d'en mesurer chacun des effets.

Comme nous l'avons vu au début de ce chapitre, l'une des méthodes d'évaluation de la précision d'un estimateur consiste à comparer la variance de l'erreur mesurée pour différents rapports signal sur bruit SNR (*Signal noise ratio*) à la courbe référentielle de *Cramer-Rao* (Cramer-rao-bound). Puisque, dans notre cas, les mesures de précision ne servent qu'à vérifier l'effet d'une quelconque modification sur le comportement de l'estimateur, nous nous contenterons, dans ce mémoire, de comparer les variances des algorithmes modifiés à celle de l'original. Ainsi, la comparaison de ces courbes nous permettra d'évaluer l'impact d'un changement sur l'architecture de l'algorithme original.

Les mesures sont faites à partir du logiciel *Matlab* de *Mathworks*. Puisque le nombre d'échantillons est un facteur déterminant de la précision, généralement, les mesures sont effectuées pour un vecteur de dimension constante de 96 échantillons. Pour des raisons issues de considérations pratiques, cette dimension de vecteur sera souvent utilisée au cours de ce mémoire pour les mesures de précision et de profilage. Il faut toutefois noter que les applications ayant recours à l'estimateur de Crozier peuvent aussi bien traiter des signaux de dimensions allant de 12 à 16 384 échantillons.

Les mesures sont faites pour des rapports SNR de –10 à 40 dB par sauts de 2.5 dB. Elles sont effectuées pour chacune des fréquences (fT) de 0.01 à 0.49 par incrémentation de 0.01.

2.3.2 Analyse de complexité

L'algorithme de Crozier est construit de quatre fonctions : estimation (DMA), décimation (RAD), Normalisation et Pondération. Les trois premières sont bien connues et facilement identifiable sur la Figure 2.6. La dernière correspond aux opérations exécutées sur les phaseurs normalisés $\hat{W}_b$ et $\hat{z}_b$ lors des changements de branches. Les équations (15),(16),(17) et (18) décrivent respectivement chacune des fonctions.

Intuitivement, on pourrait présumer que les fonctions DMA et RAD sont les plus gourmandes en termes de temps de traitement. Chacune des fonctions nécessite $4N_b$ additions (*add*) et multiplications (*mul*). Sauf exception des premières et dernières itérations, qui ne calculent que la fonction DMA, il faut, généralement, compter $8N_b$ opérations de chaque sorte pour les deux fonctions. Pour l'ensemble de l'algorithme, on a un nombre d'opérations O de

$$O = 4N(add + mul) + \left[\sum_{b=1}^{B+1} 8N_b\left(add + mul\right)\right] + 8(add + mul) \tag{20}$$

Rappelons que pour la dernière itération il ne reste que 2 échantillons à traiter. Puisque la somme sur N_b tend vers N, on peut approximer le temps requis par l'estimation avec l'équation suivante

$$O \approx 12N(add + mul) \tag{21}$$

Contrairement aux fonctions RAD et DMA, les fonctions de normalisation et de pondération ne traitent qu'une seule donnée par branche. La normalisation, dont l'équation est décrite ci-dessous, nécessite deux multiplications, une addition, deux divisions, ainsi qu'une racine carrée.

$$z = \frac{Z.réel + Z.imag}{\sqrt{(Z.réel)^2 + (Z.imag)^2}} \tag{22}$$

La complexité est bien moindre dans ce cas, mais la difficulté de traitement de certaines opérations est beaucoup plus élevée. La résolution d'une racine carrée en nombre entier, par exemple, passe généralement par des algorithmes comme le CORDIC (COordinate Rotation DIgital Computing) que nous décrirons au prochain chapitre. Sur le Pentium, l'opération consomme jusqu'à 70 cycles d'horloge de plus qu'une simple addition. En matériel, le traitement d'une racine carrée passe par l'implémentation d'un module CORDIC avec tout ce que cela comporte : contrôleurs, additionneurs, décaleurs, etc. La division est une autre opération plus coûteuse qu'une simple addition. Sur le Pentium, elle nécessite environ une trentaine de cycles d'horloge, alors qu'en matériel, on cherche carrément à éviter cette opération.

La fonction de Pondération est, quant à elle, encore plus compliquée. Selon l'équation(18), la fonction nécessite, avant tout, 12 multiplications et 6 additions pour les 3 produits de nombres complexes et, une division encore une fois sur des nombres complexes. L'opération qui est, sans aucun doute, la plus critique de cette fonction est l'extraction de la racine d_b. Afin de résoudre cette dernière, il faut employer un algorithme numérique complet non seulement complexe, mais dont la complexité est variable en fonction de la grandeur de d_b. Un exemple de solution à cette opération se trouve en ANNEXE B. Il n'est pas nécessaire d'en faire une profonde analyse afin de comprendre son coût élevé, autant en logiciel qu'en matériel. C'est une opération qu'il faut absolument éliminer.

En somme, la complexité de l'algorithme de Crozier se résume ainsi : les fonctions DMA et RAD ont un comportement antagoniste aux fonctions de Normalisation et de Pondération. Les premières traitent un grand nombre d'échantillons avec des opérations simples, alors que les secondes ne traitent qu'une seule donnée, mais avec des opérations plus complexes. L'accélération de calcul de telles fonctions passe par des solutions comme le traitement parallèle et la reformulation algorithmique. Pour les fonctions de grande complexité en termes de quantité d'opérations, seul le calcul parallèle des opérations permet d'accélérer le traitement. Pour les fonctions de grande complexité mathématique qui ne sont pas parallélisables, la seule solution possible est de réduire cette complexité en remplaçant les opérations critiques par des opérations plus simples. Pour réduire la complexité, deux options s'offrent : la reformulation algorithmique ou l'emploi de méthodes de traitement numérique avancées. Le prochain chapitre est consacré à démontrer en détail comment, à l'aide d'une reformulation algorithmique, la complexité mathématique des fonctions de Normalisation et de Pondération a été simplifiée.

2.3.3 Effort de calcul matériel

Le calcul parallèle est obtenu principalement par la mise en parallèle ou en pipeline des opérateurs et des fonctions. Le degré de parallèlisme dépend en grande partie de l'architecture matérielle de la plate-forme de traitement. À ce titre, l'avantage de l'implémentation matérielle réside dans le fait qu'il appartient au concepteur de fixer le degré de parallélisme. En logiciel, c'est l'architecture du processeur considéré qui impose au concepteur le niveau de traitement parallèle. Notons toutefois que plusieurs processeurs modernes offrent des performances de traitement très élevées. La plupart de ces processeurs exploitent, en plus d'une fréquence d'horloge très élevée, des architectures spécialement développées pour le calcul intensif.

Selon une approche matérielle, il est possible, dans le cas de l'algorithme de Crozier, de créer une structure de calcul très optimisée pour le traitement simultané des fonctions DMA et RAD. La structure décrite à la Figure 2.9 est composée de plusieurs

modules DMARAD disposés côte à côte. Le module DMARAD se compose des fonctions RAD et DMA branchées bout à bout sous une structure pipelinée. Cette architecture permet de mettre sur pied, pour chaque branche de l'algorithme, plusieurs chemins de données parallèles et pipelinés à l'intérieur desquels un flot d'échantillons circule de façon continue.

De plus, le parallélisme peut s'étendre jusqu'au calcul des multiplications et additions complexes comprises dans les modules RAD et DMA. Le multiplieur complexe est fait de 2 étages dont le premier est composé de 4 multiplieurs virgule fixe disposés en parallèle et le second, de 2 additionneurs. L'addition et l'accumulation de nombres complexes des fonctions RAD et DMA sont tous les deux faits par un seul étage de 2 additionneurs complexes placés parallèlement. Les 4 additions et multiplications nécessaires au calcul de chacune des fonctions DMA et RAD sont, ainsi, réparties selon quatre étages d'opérateurs parallèles.

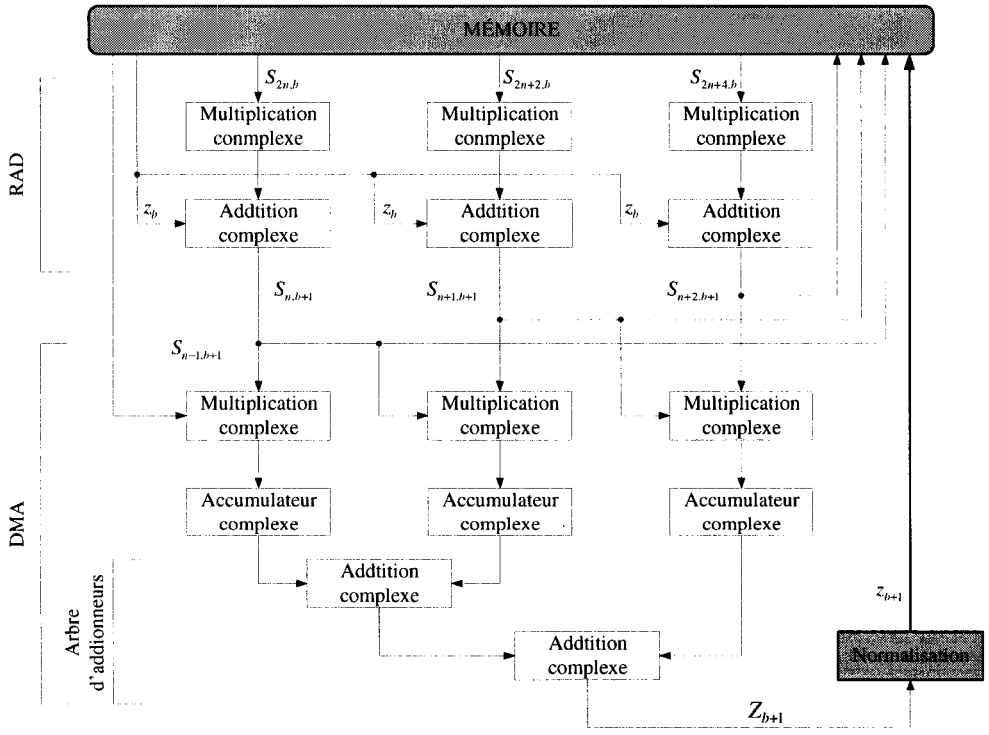

Figure 2.9 – Schéma bloc du module DMARAD

On peut estimer le délai, en nombre de cycles d'horloge, pour le calcul des fonctions DMA et RAD, sans aucune optimisation matérielle, à $8N_b$ fois la somme des délais nécessaires au calcul d'une addition et d'une multiplication soit

$$D_{DMARAD} = 8N_b(\Delta a + \Delta m) \tag{23}$$

où Δa et Δm représentent respectivement les délais combinatoires, accordés aux additionneurs et multiplieurs d'une technologie donnée. Si on utilise L modules DMARAD, où la profondeur de pipeline, K, est formé d'étages de multiplieurs et additionneurs disposés en parallèles tel que décrit ci-dessus, le délai, en nombre de cycles, devient

$$D_{DMARAD}' = \left(\frac{N_b}{L} + K\right)T_p \tag{24}$$

Où T_p représente le délai accordé à un étage de pipeline. Admettons, pour simplifier les choses, que le nombre d'étages pipeline est égal au nombre d'étages d'opérateurs et que le délai d'une multiplication est environ 3 fois celui d'une addition. On peut alors estimer K à $8+\log_2 L$, ce qui correspond à 2 étages multiplicateurs, 2 étages d'additionneurs (accumulateurs) et un arbre $\log 2L$ d'additionneurs. Si $L = 4$ et $N = 96$, on peut alors estimer le gain d'accélération de calcul à approximativement 80.

À chaque chargement du pipeline, il faut compter une latence de K cycles. Compte tenu des dépendances de données présentes entre les fonctions RAD, DMA et Normalisation (voir la Figure 2.9), le chargement du pipeline se produit à chaque itération. Cette structure sera donc plus profitable pour des itérations contenant un grand nombre d'échantillons par itération. Il est intéressant de souligner que le délai pour le calcul du bloc DMARAD est sensiblement le même que celui d'un des modules RAD ou DMA comportant les même paramètres architecturaux.

$$D_{DMA\ ou\ RAD}' = \left(\frac{N_b}{L} + \frac{K}{2}\right)T_p \tag{25}$$

Puisque pour chacun des modules on compte moitié moins d'étages d'opérateurs, la profondeur du pipeline devrait être de moitié également. Il est donc très avantageux d'utiliser la structure DMARAD aussi souvent que possible.

La dépendance de données, présente entre les modules DMARAD de branches successives, représente l'un des plus important obstacle à l'accélération de calcul. Les dépendances proviennent de l'utilisation de l'estimation normalisée pour la décimation (voir la Figure 2.9). Si cette dépendance de données pouvait être éliminée, le signal pourrait traverser de bout en bout, sans rechargement, le pipeline d'opérateurs.

Le parallélisme entre les fonctions elle mêmes est une autre façon d'accélérer le traitement. En se rapportant à la Figure 2.6, nous pouvons remarquer qu'il est possible de traiter la fonction de Pondération parallèlement au traitement des fonctions DMA-Normalisation-RAD. Considérant le parallélisme potentiel, on peut estimer le délai de l'algorithme à

$$D_{Crozier} = \Delta_{(D+N),1} + \sum_{b=2}^{B+1} \max\left[\Delta_{P,b-1} \middle| \Delta_{(DR+N),b}\right] + \max\left[\Delta_{P,B+1} \middle| \Delta_{(D+N),b+2}\right] + \Delta_{P,B+2} \quad (26)$$

où les indices *D, DR, N* et *P* font respectivement référence aux fonctions DMA, DMARAD, Normalisation et Pondération. L'indice *b* désigne l'itération. En d'autres mots, le terme $\Delta_{(DR+N),b}$ indique le délai pour le calcul successif des fonctions DMARAD et Normalisation pour l'itération *b*. La somme Σ de l'équation donne le délai pour la majorité des itérations, alors que les termes situés de part d'autre de celui-ci représentent les délais respectifs des premières et dernières itérations.

Deux simplifications peuvent immédiatement être apportées : en premier lieu, nous pouvons éliminer la dernière itération de l'algorithme. L'avantage de cette simplification est que, mis à part la première itération, toutes les itérations contiennent les mêmes fonctions à exécuter. En plus de réduire la complexité, cette simplification facilitera l'implémentation matérielle. Le coût en précision est de quelques dB seulement [19]. Pour beaucoup d'applications l'estimation demeure adéquate. Ensuite, en éliminant la dépendance de données entre les fonctions de Normalisation et de Pondération, le calcul

de cette dernière peut être amorcé plus rapidement dans la séquence. La figure suivante illustre schématiquement cette idée.

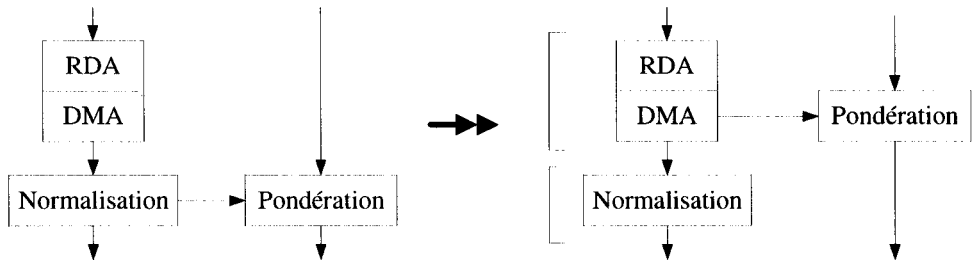

Figure 2.10 - Schémas bloc illustrant une transformation architecturale possible de l'algorithme de Crozier

L'estimation du délai devient donc

$$D_{Crozier} = \Delta_{(D),1} + \sum_{b=2}^{B+1} \max\left[\Delta_{P,b-1} \middle| \Delta_{(DR+N),b}\right] + \Delta_{P,B+1} \tag{27}$$

En débutant le calcul de la fonction de Pondération plus rapidement, on élimine un élément de latence à la dernière itération. Comme il le sera démontré au prochain chapitre, la dépendance de données est éliminée en procédant à une conversion en représentation polaire de l'estimation.

2.3.4 Effort de calcul logiciel

Puisqu'une méthode de profilage a déjà été mise sur pied, il s'avère beaucoup plus simple de mesurer directement l'effort de calcul logiciel que d'en faire l'estimation. Le tableau suivant montre des résultats de délais mesurés pour chaque fonction. Les fonctions ont été implémentées en langage C. Les opérations mathématiques proviennent toutes de la bibliothèque fournie avec le langage C. Les procédures de profilage sont les mêmes qu'à la section 2.3.1.1. Si la fonction le permet, des mesures en virgule fixe ont été également prises. Rappelons que la fonction DMARAD comprend les fonctions DMA, RAD, et Normalisation. Le nombre d'échantillons du signal est de 96. Pour la

fonction de Pondération, puisque le calcul de la racine variable d_b dépend de l'itération en cours, nous avons mesuré des délais pour les itérations de 1 et 7 (d_b=1 et 64), qui représentent les cas extrêmes pour un signal de 96 échantillons.

Première constat : la fonction de Pondération consomme de 800 à 7500 cycles selon l'itération. On constate même que pour certaines itérations, elle consomme plus de temps que les fonctions DMA, RAD et Normalisation ensemble. Il faut noter que les délais des fonctions DMA, RAD et Pondération sont tous dépendants du nombre d'échantillons du signal. Plus le signal comporte un grand nombre d'échantillons, plus la complexité des fonctions DMA/RAD augmente et plus le délai d_b augmente aussi. Le Tableau 2.3 montre des mesures prises pour les fonctions DMARAD et Pondération pour des signaux comportant de 12 à 1536 échantillons.

Tableau 2.2 – Mesures de délais pour les fonctions de l'estimateur Crozier

Fonction	Type de données (en C)	Cycles	Délai (ns)
Virgule Flottante			
DMA	Float	3800	4.4
RAD	Float	2100	2.4
Normalisation	Float	154	0.2
DAMRAD	Float	6900	8
Pondération	Float	800 à 7500	9.7
Virgule Fixe			
DMA	Integer	3400	4
RAD	Integer	8160	9.4

Le nombre d'échantillons du signal respecte, évidemment, la restriction relative au nombre d'échantillons ($N = 3x2^B$). Les résultats apportés par ces mesures montrent que

les délais pour les deux fonctions croient de façon linéaire avec le nombre d'échantillons. Nous pouvons également constater que la pente de la fonction DMARAD est beaucoup plus grande que celle de la fonction de Pondération. Bien que pour des signaux de 192 échantillons et plus, ce soit la fonction DMARAD qui domine l'effort de calcul, il n'en demeure pas moins que la fonction de Pondération contribue beaucoup au délai total de l'algorithme.

Seconde observation : le délai en virgule fixe de la fonction RAD est environ 4 fois plus grand que son délai en virgule flottante, alors que pour la fonction DMA, on observe une légère diminution en sens inverse. Ceci s'explique par le fait que la fonction RAD nécessite beaucoup d'opérations supplémentaires pour la réduction d'échelle.

Tableau 2.3 – Délai de la fonction DMARAD pour différente largeur de signal.

Nombre d'échantillons	Nombre d'itérations	DMARAD (Cycles)	Pondération (Cycles)
12	4	980	4350
24	5	1800	5350
48	6	3500	6400
96	7	6900	7500
192	8	13700	8600
384	9	27250	9700
768	10	54400	10800
1536	11	109800	11850

2.3.5 Importance la fonction RAD

Comme nous l'avons vu un peu plus haut, la fonction RAD limite grandement l'accélération de calcul. En utilisant les résultats de l'estimation pour la décimation, cette fonction génère des dépendances de données qui nuisent aux effets du pipeline et

du parallélisme. En contrepartie, cette fonction tient un grand rôle au niveau de la précision des estimations. La figure suivante démontre les comportements de l'algorithme avec et sans la fonction RAD. Pour la mesure sans RAD, une simple décimation (de 2) du signal est utilisée.

Nous pouvons remarquer sur la figure que le décalage par rapport à la courbe originale est d'une quinzaine de dB pour la majorité des rapports SNR. Toute modification concernant la simplification de cette fonction aura nécessairement de grands effets sur la justesse des estimations. Un compromis sera toutefois présenté au prochain chapitre.

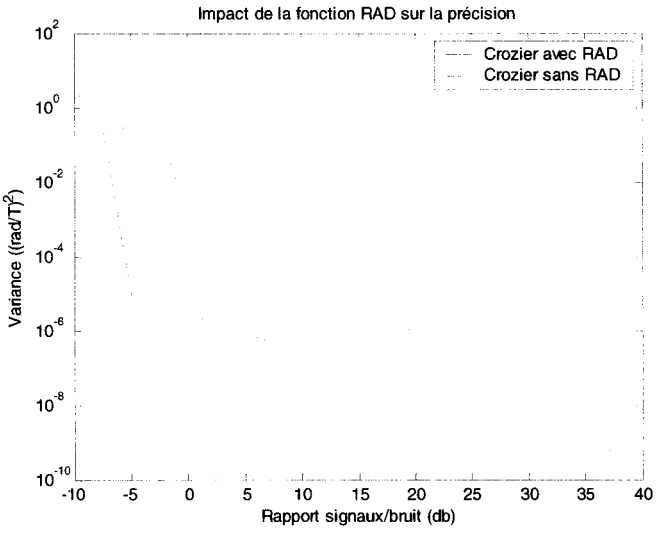

Figure 2.11 – Impact de la fonction RAD sur la précision.

2.3.6 Synthèse de l'analyse

L'algorithme de Crozier présente, au niveau de sa complexité, une problématique à caractère antagoniste intéressante: les fonctions RAD et DMA doivent traiter une grande quantité d'échantillons avec des opérations de bases telles qu'une addition ou une multiplication, alors que les fonctions de Normalisation et de Pondération n'ont, quant à elles, qu'une seule donnée à traiter, mais avec des opérations bien plus complexes. Pour

les fonctions RAD et DMA, l'accélération de calcul passe nécessairement par un traitement parallèle des données. Bien que les processeurs modernes offrent un certain degré de parallélisme, c'est la solution matérielle qui offre le plus grand potentiel de traitement parallèle.

La complexité de la fonction de Pondération a clairement été mise en évidence par les mesures de profilage. D'un point de vue de l'implémentation, c'est la fonction problématique de l'algorithme. Nous pouvons même aller plus loin en affirmant que le calcul de la racine d_b représente l'opération critique de l'algorithme. La solution à ce problème passe pratiquement par l'élimination de cette opération critique. Nous allons voir, au prochain chapitre, une reformulation algorithmique qui permet de supprimer cette opération en réduisant ainsi la complexité mathématique de la fonction de Pondération.

CHAPITRE 3
SIMPLIFICATIONS, MODIFICATIONS ET REFORMULATIONS ALGORITHMIQUES

À partir des propriétés de l'estimateur Crozier, spécifiées au chapitre précédent, nous présentons, maintenant, un ensemble de modifications, de simplifications et de reformulations pouvant conduire à des implémentations efficaces de l'algorithme.

En premier lieu, nous présenterons une reformulation algorithmique qui permet de simplifier grandement la complexité mathématique de la fonction de Pondération. Nous traiterons, ensuite, de modifications nécessaires à une implémentation, logicielle ou matérielle, en virgule fixe. Finalement, nous proposerons des variantes architecturales qui offrent un compromis au problème de dépendances de données reliant les fonctions de décimation et d'estimation.

Les effets tant sur la précision que sur la complexité ou l'effort de calcul seront analysés tout au long du chapitre.

3.1 CALCUL DE LA PHASE EN REPRÉSENTATION POLAIRE

L'un des principaux points de congestion du traitement de l'algorithme se situe à la fonction de Pondération. Nous l'avons mentionné précédemment, la complexité des opérations de cette fonction, spécialement pour le calcul d'une racine variable, exige un grand effort de calcul. Pour une bonne part, cette grande complexité découle du traitement en représentation vectorielle des nombres complexes. Crozier [19] propose une reformulation de l'algorithme qui conduit à une grande simplification mathématique : en convertissant, à chaque itération, le phaseur complexe Z_b en représentation polaire, la plupart des opérations problématiques, comprises dans la fonction de Pondération, sont remplacées par des opérations plus simples.

Globalement, les simplifications mathématiques sont facilement observables en examinant l'équation (18) qui passe de

$$\hat{W}_b = \left(\hat{z}_b \cdot \hat{z}_{b-1}^{-2}\right)^{1/d_b} * \hat{W}_{b-1}$$

à

$$\hat{F}_b = \hat{F}_{b-1} + (\hat{P}_b - 2\hat{P}_{b-1})/d_b \tag{28}$$

où $\hat{F} = \angle\hat{W}$ et $\hat{P} = \angle Z$. Ainsi, les multiplications de phaseurs complexes $\hat{W}$ et $\hat{z}$ sont réduites à de simples additions sur les phases $\hat{P}$ et $\hat{F}$. Le calcul de la racine d_b, dont l'implémentation nécessite des boucles de multiplications et d'additions de nombres complexes, est remplacée par une division que l'on peut aussitôt substituer par un décalage de $log_2 d_b$ bits. Puisque d_b est, en tout temps, une puissance de deux, il est possible de réduire la division en une suite de décalages de 1 bit vers la droite. De la même façon, la puissance (–2) appliquée à $\hat{z}_{b-1}$ est remplacée par un décalage vers la gauche de 1 bit. Le diagramme de la Figure 3.1 illustre une branche de l'algorithme de Crozier sous sa nouvelle forme. Nous pouvons remarquer que la dépendance de données entre les fonctions de Normalisation et de Pondération est désormais éliminée.

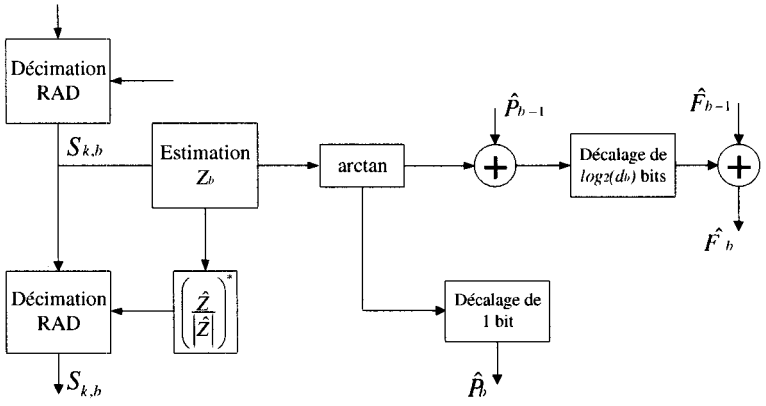

Figure 3.1 – Branche de l'algorithme de Crozier en représentation polaire

Deux inconvénients découlent de l'utilisation de la représentation polaire. D'abord, il faut convertir le phaseur complexe Z_b en représentation polaire. La conversion vectorielle à polaire se fait par l'opération arctangente, qui induit une certaine complexité. Le traitement de cette fonction, sur le Pentium, via la fonction *atan2* du langage C, peut exiger de 310 à 330 cycles d'horloge. En virgule fixe, il existe plusieurs algorithmes pour le calcul de l'arctangente. L'algorithme de CORDIC en est un qui est reconnu pour son efficacité.

Le second inconvénient relève directement du comportement périodique de la représentation polaire. Plus précisément, le problème provient de la discontinuité de l'estimation fréquentielle, sous forme de phase, autour du cercle trigonométrique. Le phénomène de discontinuité, qui n'est pas soulevé par Crozier dans sa proposition de reformulation algorithmique, a d'énormes impacts tant sur la précision de l'estimation que sur la complexité de calcul. Nous allons donc, dans les prochaines sections, voir en détail le phénomène de discontinuité, proposer des solutions et en évaluer les incidences sur la précision et l'effort de calcul.

3.1.1 Discontinuité de l'estimation autour du cercle trigonométrique

Le comportement de l'algorithme de Crozier peut être imagé par une course de deux phases autour du cercle trigonométrique; À chaque branche, l'algorithme calcule la différence entre les estimations produites aux itérations successives. La Figure 3.2 montre un exemple de l'évolution des estimations fréquentielles, sous forme de phases, pour les trois premières itérations de l'algorithme. La différence entre l'estimation courante et le double de la précédente dépend du rapport SNR.

Peu importe la méthode choisie, le calcul de la phase aura toujours, sur le cercle trigonométrique, un point de cassure. Généralement, les algorithmes de calcul numérique (*atan2* de la librairie mathématique du langage C par exemple) évaluent les angles selon les intervalles $[0,\pi]$ et $[0,-\pi]$, où le point de rupture se situe à $(\pi,-\pi)$.

L'erreur sur l'estimation survient lorsque les phases successives sont, comme le montre la Figure 3.2, de part et d'autre du point de discontinuité. Par exemple, en se référant à cette figure, si les phases $\hat{P}_{b+2}$ et $2\hat{P}_{b+1}$ ont respectivement des valeurs de –179 et de 178 degrés, la différence calculée sera de -357 au lieu de 3 degrés.

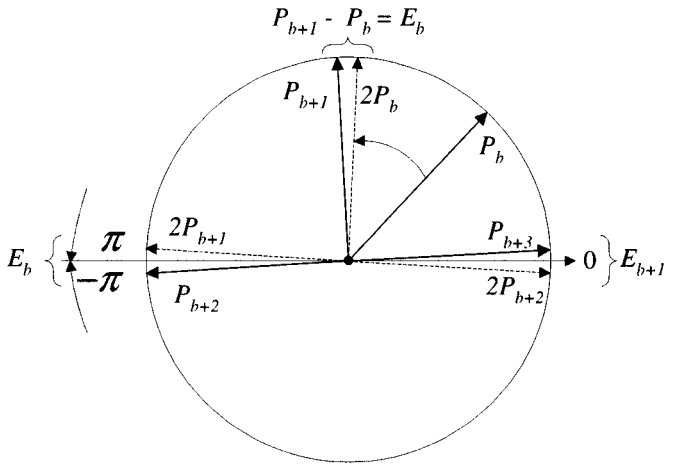

Figure 3.2 – Évolution des phases autour du cercle.

Deux situations précises peuvent entraîner ce type d'erreurs. La première, comme c'est le cas pour la Figure 3.2, survient lorsque les phases mesurées, au fil des itérations, sont proches d'angles critiques pouvant mener au point de rupture ($\pm\pi/4$, $\pm\pi/2$ et $\pm\pi$). La seconde situation se produit si le rapport SNR est assez faible pour que l'erreur E_b déplace les phases successives de chaque côté du point de rupture.

3.1.2 Détection et contrôle du cas de discontinuité

La seule façon de contrecarrer les erreurs introduites par le problème de discontinuité est de détecter les cas critiques et d'ajuster le calcul de $\hat{F}_b$ en fonction des positions des phases $\hat{P}_b$ et $\hat{P}_{b-1}$ autour du point de rupture. Pour ce faire, il suffit d'intégrer un algorithme de contrôle à la fonction de Pondération. En plus de corriger le

calcul sur $\hat{F}_b$, l'algorithme de contrôle doit ajuster le calcul de la phase $\hat{P}_{b+1}$ de l'itération subséquente. Si la phase est supérieure/inférieure à $\pm\pi/2$, le double de celle-ci sera, nécessairement dans l'intervalle $[0, \mp\pi]$. Il existe plusieurs façons d'implémenter un algorithme de contrôle qui permette de détecter les cas de discontinuité et de corriger le calcul de la fonction de Pondération. La Figure 3.3 donne le code C d'une solution possible.

```
   //P = ∠Z; F = ∠W.; pi = π.

   phase = atan2(z); // point de rupture à pi et -pi.

   // Algorithme de Contrôle
   if (iteration == 1)
      F = phase;
   else if (z.re < 0)
   {
      if (phase<0) && (P>0)
         F = ((2*pi+phase-P)/delai)+F;
      else if (phase>0) & (P<0)
         F = (-1*(2*pi+P-phase)/delai)+F;
   }
   else
      Fo = ((Phase-Pi)/Di)+Fi;

   delai = 2*delai;

   // Calcul de la phase subséquente
   if (phase) > 0) && (phase > PI/2)
      p =-2*(PI-phase);
   else if (phase) < 0) && (phase < PI/2)
      p =2*(PI+phase);
   else
      p = 2 *phase;
...
```

Figure 3.3 – Code C de la fonction de Pondération.

Généralement, le choix de l'algorithme dépend de la méthode sélectionnée pour la résolution de la fonction arctangente. Par exemple, l'algorithme de CORDIC, dans sa

forme originale, n'offre qu'un intervalle de calcul de $-\pi/2$ à $\pi/2$. Il est donc nécessaire, dans cette situation, d'adapter l'algorithme de contrôle. Peu importe l'intervalle de calcul disponible, le principe est toujours le même : en premier lieu, il faut déterminer, à partir des composantes vectorielles du phaseur Z_b et des signes des $\hat{P}_b$ et $\hat{P}_{b-1}$, dans quel quadrant les phases sont positionnées et ensuite, en fonction de leurs positions relatives, il faut ajuster le calcul de différence.

3.1.3 Impact sur la complexité et la précision

La Figure 3.4 montre une comparaison entre les estimations obtenues par le Crozier original, en représentation vectorielle de bout en bout, et celui décrit ci-dessus qui utilise la représentation polaire pour le calcul de la fonction de Pondération. Ces résultats permettent de valider, au niveau de la précision de l'estimation, l'utilisation de la représentation polaire pour le calcul de la fonction de Pondération. La figure montre également l'importance de l'algorithme de contrôle.

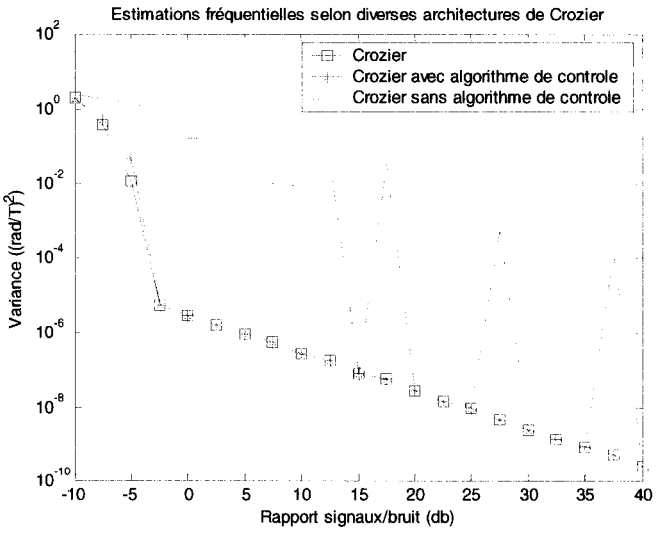

Figure 3.4 – Estimation vectorielle avec et sans algorithme de contrôle

Comme nous pouvons le constater sur cette figure, la courbe illustrant le comportement de l'algorithme de Crozier sans algorithme de contrôle diverge en plusieurs points, de la courbe de référence. Pour les rapports SNR inférieurs à 15dB, nous pouvons remarquer que la divergence est relativement constante. Pour ces rapports, comme nous l'avons expliqué précédemment, les discordances sont le résultat d'une grande portion de bruit dans le signal. Pour les trois autres points de divergence, nous pouvons présumer que des erreurs ont été malencontreusement induites par l'apparition d'angles critiques dans le processus de résolution.

En ne comparant que les équations (18) et (28), il est évident que la complexité de la fonction de Pondération est énormément réduite par l'utilisation de la représentation polaire. D'une part, les multiplications complexes deviennent de simples additions et, d'autre part, la fonction itérative de la racine variable est remplacée par une opération de décalage à délai fixe. Il faut toutefois considérer l'apport dû à la fonction de conversion vectorielle-polaire. Nous l'avons mentionné un peu plus haut, le calcul de l'arctangente sur un Pentium consomme jusqu'à 330 cycles d'horloge. En virgule fixe, via le CORDIC, on peut compter de 350 à 500 cycles selon le nombre d'itérations employées. Du point de vue matériel, la simplification est encore plus évidente. Le simple fait d'éliminer une fonction itérative et complexe comme celle de la racine variable représente, en soi, une grande simplification de la réalisation matérielle. De plus, comme nous le verrons au CHAPITRE 5, le traitement parallèle qu'offre la solution matérielle permettra de créer une fonction très efficace qui sera traitée en quelques cycles d'horloge.

Il est plus facile, encore une fois, d'évaluer le gain global attribuable à la reformulation algorithmique par des mesures logicielles. Le tableau suivant compare les performances de la fonction de Pondération obtenues pour des implémentations en représentation polaire et vectorielle. Pour la version vectorielle, nous reportons ici les résultats démontrés au Tableau 2.3. Dans les deux cas, nous avons fait des mesures pour les délais associés aux itérations 1 et 11 (d_b=1 et 1024), qui représentent les cas extrêmes pour un signal de 1536 échantillons (rappelons que $d_b=2^{b-1}$ pour $b=1,2,..,B+1$ et

$B=log_2(N/3)$). Les résultats montrent que le délai, en cycles de l'horloge, de la fonction de Pondération en mode polaire est indépendant du nombre d'échantillons, ce qui représente un grand avantage. De plus, selon l'itération en cours, la version polaire est de 2 à 30 fois plus rapide que la version vectorielle

Tableau 3.1 - Comparaison entre représentation rectangulaire et polaire.

Représentation	Format des données	Type de données	Cycles	Délai (ns)
Vectorielle	réels	Float	800 à 11850	0.92 à 13.6
Polaire	réels	Float	400	0.46

3.2 TRAITEMENT EN VIRGULE FIXE

3.2.1 Algorithme de pseudo Normalisation

Généralement, les opérations de division et de racine carrée sont coûteuses en ressources, peu importe la forme de l'implémentation. Sur le Pentium, les mesures de profilage montrent que ces opérations demandent respectivement au moins 30 et 80 cycles d'horloge. En nombre entier, l'implémentation d'une racine carrée implique obligatoirement un algorithme numérique comme le CORDIC. En matériel, la division est simplement une opération que tout concepteur tente d'éviter. Bref, s'il est possible de le faire, vaut mieux éliminer ces opérations du traitement.

Une alternative au calcul de la Normalisation consiste à obtenir, par le biais de décalages binaires successifs, une réduction d'échelle qui donne un résultat approximativement équivalent à celle-ci.

Selon la reformulation proposé à la section 3.1, le phaseur normalisé de $\hat{z}_b$ n'est utilisée que par la fonction RAD. Rappelons que le rôle du phaseur normalisé et conjugué $\hat{z}_b^*$ est d'appliquer une correction vectorielle à l'un des deux échantillons

impliqués dans la décimation RAD. Les Figure 2.4 et Figure 2.5 du CHAPITRE 2 montrent le comportement angulaire de la fonction RAD et le rôle qu'y joue la Normalisation. Il est évident que l'utilisation d'une approximation de la norme, dans le calcul du vecteur normalisé $\hat{z}_h$, modifiera la longueur de ce dernier qui, à son tour, modifiera l'angle des échantillons résultants de la décimation et, éventuellement, l'estimation de la fréquence (voir la Figure 2.5). On peut facilement présumer qu'il faut une différence importante entre l'approximation et la norme elle-même afin que la répercussion sur l'estimation de la fréquence soit significative.

La figure suivante montre l'algorithme, basé sur les décalages successifs, qui permet de calculer efficacement une approximation du vecteur normalisé. Le principe consiste à diviser par 2 chacune des composantes vectorielles du phaseur Z_h, issues de la fonction DMA, jusqu'à ce que la plus grande des deux, en valeur absolue, soit inférieure à 1. L'algorithme porte le nom de Pseudo Normalisation.

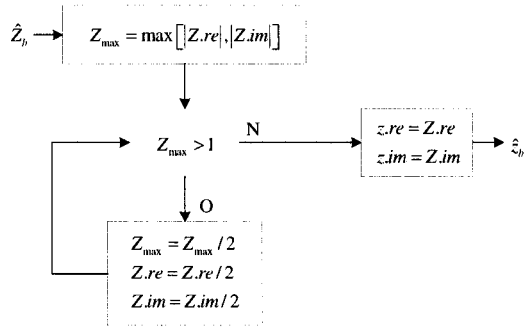

Figure 3.5 – Algorithme de pseudo normalisation

L'approximation rendue par cet algorithme est adéquate tant et aussi longtemps que la norme du phaseur Z_h est supérieure à 1/2. Normalement, l'amplitude des signaux analysés garantit cette hypothèse. Il est toutefois facile de corriger l'algorithme en fonction de cette restriction si les signaux ont une amplitude relativement faible.

La norme du vecteur approximatif peut varier de $1/\sqrt{2}$ à $\sqrt{2}$. La différence entre le vecteur réduit et la norme est, ainsi, assez petite pour ne pas trop influencer la phase de l'échantillon produit par la décimation. La Figure 3.6 montre l'effet de la Pseudo Normalisation sur la précision de l'estimateur. On peut compter une différence 1 à 3 dB entre les courbes du Crozier original et du Crozier modifié, ce qui est très acceptable.

En virgule fixe, l'économie du temps de calcul logiciel découle en grande partie de l'élimination de la racine carrée, ce qui permet de supprimer plusieurs centaines de cycles d'horloge au délai. Bien que les divisions (30-40 cycles) soient remplacées par des opérations de décalage (5 cycles), puisque l'algorithme peut exécuter plusieurs itérations, l'économie apportée par cette dernière peut vite s'amenuiser. Sur le Pentium, l'algorithme de Pseudo Normalisation décrit à la Figure 3.6 peut consommer de 140 à 220 cycles d'horloge pour un nombre d'itérations allant de 1 à 8. Il faut toutefois souligner que l'utilisation d'instructions en langage assembleur, propre à la machine considérée, pourrait réduire d'avantage le temps de calcul.

En nombres réels, le calcul de la fonction de Normalisation à l'aide de la bibliothèque mathématique du langage C ne nécessite que 154 cycles. Le coût n'est pas assez grand pour justifier l'emploi de l'algorithme de Pseudo Normalisation.

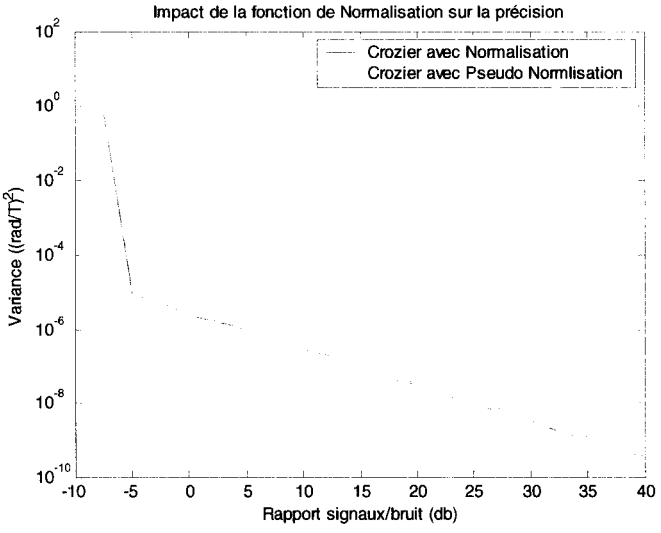

Figure 3.6 – Impact de la fonction Normalisation sur la précision.

En matériel, comme nous le verrons au CHAPITRE 5, l'algorithme de Pseudo Normalisation s'exécute en moins de 5 cycles d'horloge. De plus, à cause du parallélisme qu'offre la solution matérielle, le délai de l'algorithme est indépendant du nombre d'itérations. La performance matérielle du Pseudo Normalisateur est donc énormément avantageuse.

3.2.2 Conversion cartésienne-polaire : CORDIC

Nous l'avons mentionné à plusieurs reprises, l'algorithme de CORDIC (COordinate Rotation DIgital Computing) [2] représente une solution simple et efficace à la résolution numérique d'opérations complexes comme la racine carrée et l'arctangente. S'appuyant sur la rotation vectorielle, l'algorithme calcule, à l'aide de simples opérations d'additions et de décalages, la plupart des fonctions trigonométriques. Originellement conçu pour des problèmes de navigation, l'algorithme de CORDIC se trouve, aujourd'hui, utilisé dans de nombreuses applications commerciales : calculatrices, robotiques ou systèmes de communication numériques de toutes sortes. Dans notre cas, il sera utilisé afin de produire rapidement des conversions cartésiennes-polaires sur des échantillons d'un signal.

L'algorithme de CORDIC résout les fonctions trigonométriques en effectuant une série de rotations vectorielles dans le plan cartésien. Considérons le vecteur $V(x,y)$, auquel une rotation de φ est appliquée. Les coordonnées du vecteur V', résultant de la rotation, sont exprimées selon les équations

$$\begin{aligned} x' &= x\cos(\varphi) - y\sin(\varphi) \\ y' &= y\cos(\varphi) + x\sin(\varphi) \end{aligned} \tag{29}$$

que l'on peut réorganiser sous la forme

$$\begin{aligned} x' &= \cos(\varphi)\left[x - y\tan(\varphi)\right] \\ y' &= \cos(\varphi)\left[y + x\tan(\varphi)\right] \end{aligned} \tag{30}$$

Une façon d'obtenir une rotation de φ consiste à effectuer, selon un processus itératif, une série de rotations φ_i égales à $tan^{-1}(\pm 2^{-i})$ où $i = 0,1,2,3,....$ L'angle a_i pour chaque itération du processus est

$$a_{i+1} = a_i - d_i \tan^{-1}(2^{-i}) \tag{31}$$

où d_i, égal à ± 1, indique le sens de la rotation à appliquer à chaque itération. Les composantes x_{i+1} et y_{i+1} du vecteur itératif V_{i+1} sont obtenues par

$$\begin{aligned} x_{i+1} &= K_i \left[x_i - y_i d_i 2^i \right] \\ y_{i+1} &= K_i \left[y_i + x_i d_i 2^i \right] \end{aligned} \tag{32}$$

où $K_i = \cos(\tan^{-i} 2^i) = (1+2^{-2i})^{-1/2}$. Notons que les multiplications impliquant la tangente, $tan(\varphi)$, dans les équations (30) sont remplacées par de simples opérations de décalage de 1 bit. Pour plusieurs itérations, les facteurs K_i successifs peuvent être mis en évidence. Pour un nombre d'itérations élevé, le produit des facteurs K_i tend vers la valeur de 0.6073. Le gain, A_n, de l'algorithme de rotation est donc approximativement de 1.647. Ainsi pour n itérations on a

$$A_n = \prod_n \sqrt{1+2^{-2i}} \tag{33}$$

Puisque pour un nombre relativement élevé d'itérations, le produit tend vers un résultat constant, il nous est possible de retirer ce produit des équations présentées ci-dessus et de l'appliquer ailleurs. On obtient, ainsi, un ensemble d'équations qui définissent la rotation vectorielle selon une séquence de rotations élémentaires.

$$\begin{aligned} x_i &= x_i - d_i y_i 2^{-i} \\ y_i &= y_i - d_i x_i 2^{-i} \\ a_i &= a_i - d_i \tan^{-1}(2^{-i}) \end{aligned} \tag{34}$$

Le principe de l'algorithme consiste à faire pivoter, par un angle de plus en plus petit, dans le sens approprié, le vecteur de rotation jusqu'à ce que l'angle a_i ou les composantes x_i et y_i, selon le mode d'opération, soient approximativement égales à 0.

Les composantes différentes de 0 donnent les résultats des opérations sinus, cosinus, racine carrée et arctangente. Il existe deux modes d'opération possibles : rotation et vecteur. Le Tableau 3.2 donne les caractéristiques relatives à chacun des modes d'opérations.

Tableau 3.2 – Propriétés de l'algorithme CORDIC.

Fonction	Mode	x_0	y_0	a_0	$d_i=-1$	Réponse
Sin(A)	Rotation	$1/A_n$	*0*	*A*	*a<0*	y_n
Cos(A)	Rotation	$1/A_n$	*0*	*A*	*a<0*	x_n
Atan (Y/X)	Vecteur	*X*	*Y*	*0*	*y>0*	a_n
$\frac{A_n}{\sqrt{X^2+Y^2}}$	Vecteur	*X*	*Y*	*0*	*y>0*	$x_n \cdot A_n$

Selon le premier mode, la composante a_0 est initialisée à un angle donné par l'usager alors que x_0 et y_0 portent respectivement les valeurs $1/A_n$ et 0. En ramenant l'angle a_0 à 0, les composantes x_0 et y_0 prennent respectivement les grandeurs du sinus et du cosinus de ce dernier. Le sens de rotation d_i, dans ce cas, est déterminé par l'angle a_i.

En mode vecteur, l'usager fixe les grandeurs des composantes x_0 et y_0 alors que a_0 est initialisé à 0. En ramenant la composante y_i à 0, a_i prend la valeur de l'angle décrit par les composantes x_0 et y_0. La composante x_i, quant à elle, donne la norme du vecteur. Le sens de rotation d_i est déterminé, pour ce mode d'opération, par l'angle y_i.

L'implémentation logicielle de cet algorithme est relativement simple. Il ne nécessite que des opérations d'additions et de décalages binaires. L'ANNEXE C contient un exemple de code montrant une implémentation logicielle. Sur le Pentium, le CORDIC peut prendre jusqu'à 500 cycles d'horloge pour une quinzaine d'itérations. L'implémentation matérielle du CORDIC est légèrement plus compliquée. Le sujet sera

amplement discuté au CHAPITRE 5. Mentionnons seulement pour l'instant qu'il faut à peu près 20 cycles pour un processus de 15 itérations sur une implémentation matérielle.

3.2.3 Impact sur la précision

Il serait légitime, à ce stade, de vouloir vérifier l'effet de toutes les modifications, apportées jusqu'à maintenant, sur la précision de l'estimation fréquentielle. Théoriquement, la reformulation algorithmique, présentée à la section 3.1, ainsi que l'utilisation du CORDIC, pour la conversion cartésienne-polaire, ne devraient pas influencer la justesse des estimations. La figure suivante montre, en effet, qu'il y a très peu de différence entre la courbe originale et celle de l'algorithme modifié. L'algorithme modifié comprend : la reformulation algorithmique pour le calcul de la fonction de Pondération en représentation polaire, le CORDIC pour la conversion cartésienne-polaire et l'algorithme de Pseudo Normalisation. Comme prévu, la différence entre les courbes est attribuable uniquement à l'algorithme de Pseudo Normalisation.

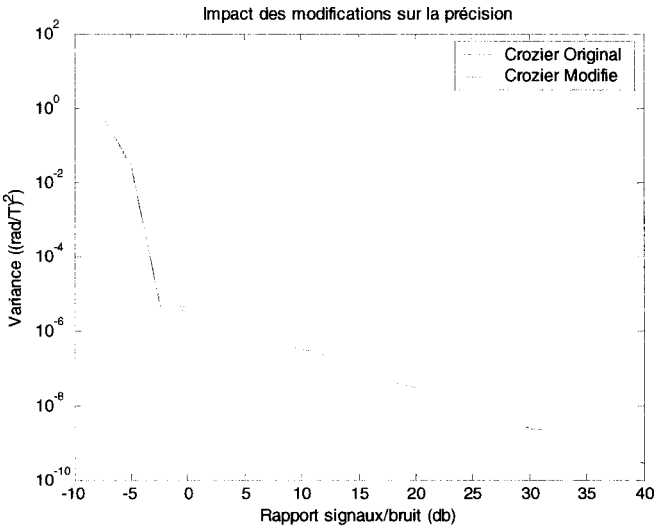

Figure 3.7 – Effets des modifications sur la précision.

Le seul paramètre restant inconnu pour l'implémentation matérielle en virgule fixe est la largeur des mots à utiliser : ce qui est non seulement déterminant pour la vitesse du traitement et la surface requise, mais aussi pour la précision de l'estimation. Il est toutefois important de mentionner qu'un paramètre aussi significatif pour la performance d'un algorithme relève d'une problématique tout aussi importante et percutante. La résolution de cette problématique est très complexe et dépasse le cadre de ce mémoire. Le sujet est d'ailleurs largement débattu au cœur de la communauté scientifique. Il existe des méthodes et outils qui ont pour fonction de déterminer la dimension optimale des mots selon une précision de calcul désirée. L'un d'eux, développé par Marc-André Cantin [3], a été utilisé pour analyser les algorithmes de CORDIC et de CROZIER.

3.3 VARIANTES ARCHITECTURALES

Bien que la fonction de décimation RAD joue un rôle crucial dans la précision de l'estimation, il est intéressant d'envisager des alternatives qui permettent de contourner, par le biais de compromis, les restrictions qu'elle impose à l'accélération de calcul. Rappelons qu'à chaque itération, le pipeline du module DMARAD doit être vidé et rechargé en raison de la dépendance de données existant entre les fonctions DMA et RAD. Les solutions proposées sont uniquement utiles pour une solution matérielle.

Une première alternative consiste à utiliser la première estimation soit, Z_1, pour toutes les décimations produites à chacune des itérations. Puisque la phase estimée doit, normalement, doubler à chaque itération, il est facile de calculer préalablement, à partir de la première, une approximation de chacune des phases produites à chacune des branches de l'algorithme. Ainsi, la dépendance de données reliant les fonctions DMA, Normalisation et RAD est, désormais, brisée pour toutes les itérations sauf la première. Sans en faire une analyse exhaustive, on peut facilement admettre qu'une approximation de la phase estimée peut facilement faire l'affaire pour le calcul de l'opération de décimation. Néanmoins, cette technique, que nous nommerons estimation unique, est

basée sur l'hypothèse que la phase doit approximativement doubler à chaque itération, ce qui est vrai pour des rapports signal sur bruit assez élevés. Pour de faibles SNR, rien ne garantit que la phase estimée doublera de branche en branche.

Une autre alternative consiste à produire des estimations de phases partielles parallèlement à celle produite par le module DMARAD. Les estimations partielles sont faites à partir d'un plus court segment du signal. Les phases issues de ces estimations partielles sont utilisées pour les opérations de décimation. Le but visé par cette modification est d'éliminer la latence du remplissage attribuable au pipeline que l'on retrouve à chaque itération. Par exemple, en utilisant un signal de $N_b/3$ échantillons pour estimer le phaseur partiel $Z_b^{'}$, il est possible d'amorcer la décimation de l'itération suivante, $b+1$, sans avoir à vider le pipeline du module DMARAD. La Figure 3.8 montre l'architecture du module basée sur l'estimation partielle.

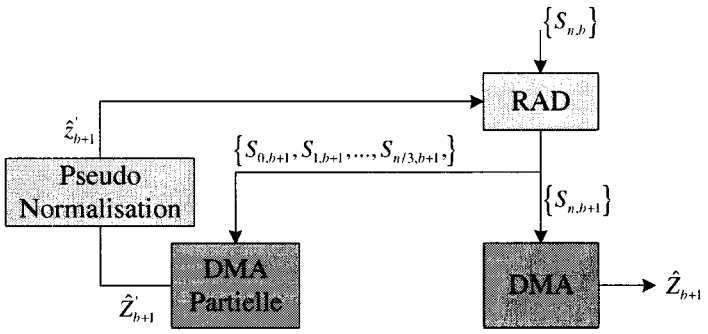

Figure 3.8 – Module DMARAD basé sur l'estimation partielle.

L'avantage de cette architecture est de pouvoir, dans la majorité des cas, éliminer du délai la latence du pipeline. Elle sera, ainsi, très avantageuse pour des signaux dont le nombre d'échantillons est petit. À partir de l'équation (24), l'estimation du délai devient

$$D_{DMARAD}{}'' = \frac{N_b}{L} T_p \tag{35}$$

La figure suivante montre l'effet des compromis proposés sur la précision de l'estimation. On constate que le traitement basé sur l'estimation partielle atteint la courbe du Crozier original à un seuil SNR plus bas que celui basé sur l'estimation unique. L'utilisation de l'estimation partielle a l'avantage de mieux respecter le comportement du bruit sur le signal. Puisque les estimations utilisées pour la décimation sont obtenues qu'à partir du signal de l'itération courante, plutôt que du signal initial uniquement, l'effet du bruit que l'on rencontre dans chaque signal sera aussi visible dans l'estimation obtenue de ceux-ci.

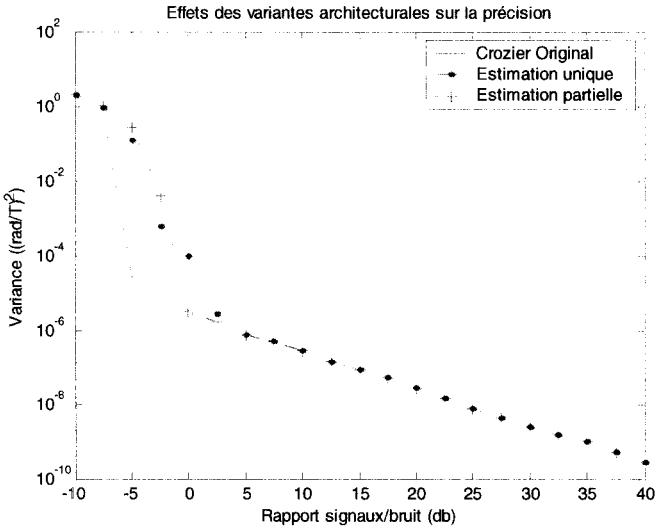

Figure 3.9 – Effets des variantes architecturales sur la précision.

CHAPITRE 4
IMPLÉMENTATION LOGICIELLE

Aujourd'hui, presque tous les ordinateurs de bureau utilisent l'un des processeurs bien connus de l'une des trois grandes familles, soit Pentium (Intel), Athlon (AMD) ou PowerPC (Motorolla/IBM). Pendant plusieurs années, ces processeurs que l'on qualifiait de processeurs à usage générique se partageaient les parts de marché du multi usage : bureautique, multimédia, etc. Toutefois, depuis quelques années, on les voit prendre place dans des marchés plus pointus où le traitement de données et la puissance de calcul ont préséance sur la multi fonctionnalité. Des domaines comme les communications numériques, par exemple, faisaient, pendant longtemps, usage de processeurs spécialisés pour le traitement des signaux (communément appelés *DSP-Digital signal processors*) spécialement adaptés aux besoins de ces classes d'application. Aujourd'hui, il est fréquent de voir des plates-formes de traitement de signal numérique construites de plusieurs Pentiums ou PowerPC disposés en parallèles.

On peut noter trois facteurs dominants qui ont amené ces machines au rang des processeurs de hautes performances. D'abord, l'apparition de techniques de calcul avancées comme la technologie MMX (*Multimedia Extensions*) des Pentiums ou AltiVec de PowerPC, ont permis à ces processeurs d'atteindre des performances très compétitives. Ensuite, en raison de la grandeur et de la variété des marchés auxquels ces processeurs s'adressent, les compagnies Intel, AMD et IBM/Motorola, ayant les ressources nécessaires, ont su non seulement développer des architectures très performantes mais également des méthodes d'implémentations efficaces. Finalement, la très grande popularité de ces processeurs favorise, d'une part, un meilleur support au développeur et, d'autre part, une plus grande portabilité. Le terme support fait, ici, référence à tout ce qui est support technique, échange d'information, système d'exploitation en temps réel, etc.

Il serait légitime, à ce stade, de vouloir comparer les différents produits de chacune des familles. La question pourrait être débattue longtemps sans jamais réellement y trouver de gagnant. Étant donné la très forte compétition qui règne au sein de ce marché, les performances de chacune de ces familles de produits sont comparables. Ne serait-ce que pour sa grande popularité et, surtout, son accessibilité, notre choix s'est arrêté sur la famille des Pentiums. Rappelons toutefois que l'objectif de ce travail est d'évaluer une méthode de conception et non les performances propres à une machine spécifique.

4.1 LES PENTIUMS

La haute performance des architectures 32 bits de Pentiums pour le traitement de signal (IA-32, *Intel 32 bit architecture*) est due notamment à l'apparition, au milieu des années 90, d'organes SIMD (*Single Instructions Multiple Data*) intégrés. Cette technique, qui n'est pas nouvelle, permet d'exécuter, simultanément, plusieurs fois la même opération sur un ensemble d'opérandes donné. Ces derniers sont retenus à l'intérieur de registres dont la dimension est spécialement adaptée à cet effet. Les premiers processeurs largement répandus à exploiter ce concept sont les Pentiums II et MMX qui offrirent, pour la première fois, un accès facile et économique à la technique SIMD. Cette innovation a pris le nom de technologie MMX (*MultiMedia eXtensions*) et elle comprenait :

- les dispositions architecturales nécessaires au calcul parallèle via des instructions SIMD (arithmétique saturé).
- 8 registres de 64 bits.
- 4 nouveaux formats de données.
- une extension de 57 nouvelles instructions.

Les Pentiums III et IV enchaînèrent avec leurs propres extensions : SSE1 et SSE2 (*Streaming SIMD extensions 1 et 2*). Ces dernières enrichissent des propriétés déjà implantées sous la technologie MMX : plus de types de données, registres de plus grandes dimensions (de 64 à 128 bits), instructions supplémentaires.

Les trois extensions ont en commun de faire usage de la technologie SIMD et d'utiliser des registres de grandes dimensions. Par contre, les types de données visées par chacune d'elles sont assez différents. L'extension MMX est, principalement, dédiée au calcul parallèle de nombres entiers. Avec les registres de 64 bits, on peut simultanément travailler jusqu'à 4 nombres de 16 bits. L'extension SSE1 est, quant à elle, principalement dédiée au calcul parallèle de nombres réels (virgule flottante). Les registres de 128 bits peuvent contenir 4 nombres de 32 bits (*single precision* – SP) en virgule flottante. Le dernier ensemble d'instructions, que l'on retrouve sur les P4, vise autant le calcul de nombres entiers que de nombres réels. Il peut, d'une part, traiter simultanément jusqu'à 16 opérandes entiers de 8 bits et, à l'autre extrême, 2 opérandes réels de 64 bits (*double precision* – DB) en passant par divers formats intermédiaires.

4.1.1 Opérations saturées et scalaires

Les trois extensions supportent autant les opérations saturées que scalaires. Les opérations scalaires agissent sur l'élément le moins significatif de chacun des opérandes alors que les opérations saturées agissent simultanément sur les différents éléments des opérandes (Figure 4.1). Le degré de parallélisme, en mode saturé, dépend de l'instruction et du type de données en cause. Il est évident que l'accélération de calcul optimale passe par l'utilisation des opérations saturées.

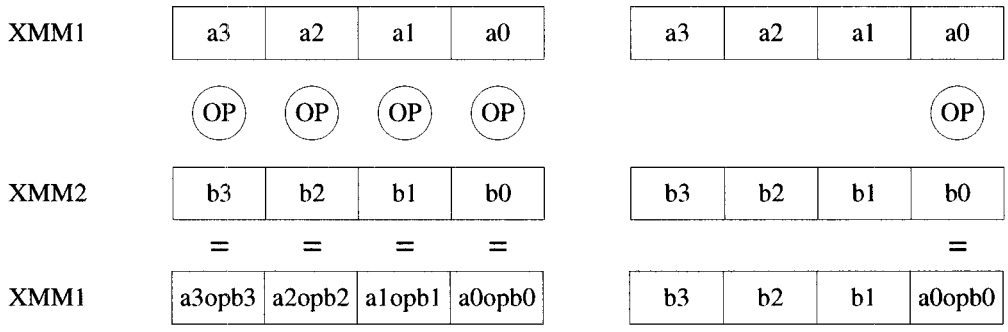

Figure 4.1 – Opération saturée (à gauche) et scalaire (à droite)

4.1.2 Types de données et registres

Les extensions utilisent différents formats de données. Le type de données __m64 est un format de 64 bits utilisé pour représenter le contenu d'un des 8 registres MM de la technologie MMX. On accède les registres XMM des extensions SSE1 SSE2 par les types de données de 128 bits __m128, __128d et __128i. Le Tableau 4.1 résume les formats de segmentation possibles pour chacun des types de données.

Tableau 4.1 – Types de données utilisés par les extensions MMX, SSE1 et SSE2

Type de données	Technologie	Registres	Formats Possibles
__m64	MMX	MM	8 entiers de 8 bits 4 entiers de 16 bits 2 entiers de 32 bits 1 entier de 64 bits
__m128	SSE1	XMM	4 réels 32 bits (SP)
__m128d	SSE2	XMM	2 réels de 64 bits (DP)
__m128i	SSE2	XMM	16 entiers de 8 bits 8 entiers de 16 bits 4 entiers de 32 bits 2 entiers de 64 bits

4.2 IMPLÉMENTATION DES EXTENSIONS MMX, SSE1, SSE2

4.2.1 Langage assembleur

L'implémentation en langage assembleur par l'entremise des extensions MMX, SSE1 et SSE2, représente la méthode la plus directe de mettre à profit la technologie SIMD. C'est également la meilleure façon d'obtenir des résultats optimaux tant au niveau de la performance que de la densité de code. Cette méthode s'avère toutefois très

complexe et elle est, généralement, laissée aux connaisseurs expérimentés des architectures IA-32 et de leurs instructions. En plus d'une expertise spécifique, la programmation en assembleur nécessite un environnement de conception qui permet d'écrire, de compiler, de simuler et de déboguer le code. Bien que les coûts engendrés par l'achat d'un tel environnent soit raisonnable (500-1500$ par licence), il n'en demeure pas moins que cette méthode de programmation limite la portabilité autant de l'application que du développement; le langage assembleur ne peut être aussi générique et portable qu'un langage de haut niveau tel le C ou le C++.

Une façon de simplifier légèrement la programmation en assembleur et d'augmenter la portabilité consiste à utiliser l'insertion de routines en assembleur dans des codes sources en C ou C++. La technique, bien connue, est communément appelée « *inline assembly* ». Rappelons simplement que le code source de l'application en cause est composé d'une trame principale en C/C++ dans laquelle on insère des portions de langage en assembleur sous forme de routines. Un exemple d'« *inline assembly* » est illustré à la Figure 4.2. La routine utilise l'instruction *addps* de l'extension SSE afin de produire une addition saturée de 4 nombres réels. Les instructions de chacune des extensions MMX, SSE1 et SSE2 ainsi que l'ensemble du jeu d'instructions des architectures IA-32 sont énumérés et décrits aux références [6][7].

```
...
   // Routine en assembleur
   __asm
   {
      push esi;
      push edi;

      mov    xmm0, xmm1
      addps     xmm0,xmm2

      pop edi;
      pop esi;
   }
   ...
```

Figure 4.2 - Exemple de programmation assembleur par la méthode « inline assembly »

Bien que ce ne soit pas le cas pour tous, certains compilateurs standards de C/C++ supportent, souvent à l'aide de mises à jour ou de macros, les routines en assembleur des machines IA-32 ainsi que les instructions MMX, SSE1 et SSE2. L'environnement de développement VISUAL C++ V6.0 de Microsoft en est un qui, selon notre expérience, supporte très bien ce mécanisme de programmation.

Sous cette forme de programmation on utilise, généralement, le langage assembleur pour le calcul d'opérations ou de fonctions isolées et précises, telle qu'une multiplication matricielle. De cette façon, plusieurs considérations relatives à la gestion du processeur, tels que les modes d'opérations, les interruptions, les exceptions, la gestion de boucles et les appels de procédures peuvent être pris en charge par un langage de plus haut niveau et son compilateur. Ces considérations deviennent ainsi transparentes pour le programmeur et lui permettent de travailler avec les extensions optimisées sans qu'il soit, pour autant, un expert des architectures IA-32. La portion d'assembleur sur l'ensemble du code peut être très variable selon les besoins et les connaissances du programmeur. Habituellement plus la portion en assembleur est élevée, plus le code tend à devenir optimisé.

Ce mécanisme de programmation reste malgré tout assez risqué. Sans une bonne connaissance de l'assembleur et du compilateur, un code très charnu en langage machine peut donner de piètres performances. Il est fortement suggéré de restreindre au minimum la portion assembleur afin de ne traiter que des opérations précises. Pour les opérations non critiques, le programmeur devrait envisager un autre mécanisme de programmation.

4.2.2 Bibliothèque Intrinsèque et classes C++ SIMD d'Intel

Deux mécanismes de programmation ont été développés par la compagnie Intel afin de faciliter l'accès aux instructions des extensions MMX, SSE1 et SSE2. Les mécanismes agissent en tant qu'interfaces qui, situées à différents niveaux d'abstraction, permettent aux développeurs d'accéder plus facilement aux extensions optimisées. La première interface est une bibliothèque en C nommée *intrinsèque*, alors que la seconde apparaît sous forme de classes C++.

4.2.2.1 Bibliothèque Intrinsèque

La bibliothèque *intrinsèque* est une interface de programmation aux instructions SIMD. Les *intrinsèques* qui composent cette bibliothèque sont des appels de fonctions en C qui intègrent, lors de la compilation, une ou des instructions en assembleur associées aux extensions. La plupart des instructions SIMD ont une *intrinsèque* correspondante; par exemple, l'instruction de l'extension SSE1 *addps*, qui réalise une addition saturée (voir Figure 4.2), correspondante à *_mm_add_ps* de la bibliothèque *intrinsèque*. Tel que décrit au Tableau 4.1, cette dernière, comme la majorité des fonctions de la bibliothèque, utilise comme arguments des types de données jumelés à son extension.

L'utilisation de la bibliothèque *intrinsèque* facilite l'intégration des extensions optimisées en évitant au programmeur, d'une part, d'avoir à utiliser le langage assembleur et, d'autre part, d'assurer la gestion des registres. De plus, avec la bibliothèque intrinsèque, c'est le compilateur et non le programmeur qui assure l'ordonnancement des instructions, l'allocation et la gestion des registres. Il arrive que dans certains cas, le compilateur puisse optimiser le code d'aussi bonne façon, sinon meilleure, que ne le pourrait un programmeur via l'assembleur [26][10]. Avec ce mécanisme de programmation, le compilateur joue un rôle important sur la performance du code.

Pour utiliser la bibliothèque *intrinsèque*, il suffit d'inclure l'un des fichiers « mmintrin.h », «xmmintrin.h » ou « emmintrin.h » qui se réfèrent respectivement aux extensions MMX, SSE1 et SSE2. La Figure 4.3 montre un exemple de programmation mettant à profit les fonctions *intrinsèques* de l'extension SSE1. Le code décrit une addition saturée de 4 nombres réels.

Les fonctions *_mm_set_ps* et *_mm_store_ps* permettent respectivement de lire et d'écrire les variables de type __m128. La fonction *_mm_add_ps*, tel que mentionné plus haut, produit une addition saturée sur deux variables de type __m128 contenant 4

nombres réels de 32 bits. Le suffixe *_ps* désigne une instruction saturée. L'ensemble des *intrinsèques* sont décrites et listées aux références [8][9][10].

```
#include <xmmintrin.h>
...
float *r;
__m128  a,b,c;
...
a = _mm_set_ps(4,3,2,1);
b = _mm_set_ps(8,7,6,5);
c = _mm_add_ps(a,b);
...
_mm_store_ps(r,c);
...
```

Figure 4.3 – Code réalisé avec les fonctions *intrinsèques*.

4.2.2.2 Classe C++ SIMD

De façon similaire au mécanisme précédent, les classes C++ d'Intel constituent aussi une interface de programmation aux instructions SIMD. Les classes implémentent les fonctions *intrinsèques* afin de définir différentes *méthodes* (fonctions membres) pour les types de données de chacune des extensions MMX, SSE1 et SSE2 (voir le Tableau 4.1). Les fonctions membres sont, pour la plupart, des surcharges des opérateurs standard d'addition, de soustraction, de multiplication, etc. Alors que la bibliothèque intrinsèque offre des équivalentes en C aux instructions en assembleur, la classe C++ SIMD permet d'utiliser directement les opérateurs standard. La figure suivante illustre un exemple de code utilisant la classe *Is16vec8* appartenant à l'extension SSE2. Le code réalise une multiplication saturée sur les objets *a* et *b* de la classe *Is16vec8.*

Cette classe définit, entre autres, les méthodes pour les opérations arithmétiques, logiques, d'affectations et de décalages pour des vecteurs de 8 entiers (*integer*) signés de 16 bits.

On distingue deux types de classes qui s'appliquent respectivement aux vecteurs de types entiers et réels: les classes *Ivec* (*Integer vector class*) et les classes *Fvec* (*Flaoting-point vector class*). Les prototypes des différentes classes sont décrits dans les fichiers d'en-tête « ivec.h », « fvec.h » et « dvec.h » qui font respectivement référence aux extensions MMX, SSE1 et SSE2. Le Tableau 4.2 énumère quelques-unes des classes offertes par les librairies d'Intel. Les propriétés et les prototypes des classes sont entièrement décrits à la référence [10].

```
#include <dvec.h>
....
int a1,b1,c1,d1,e1,f1,g1,h1;
int a2,b2,c2,d2,e2,f2,g2,h2;

Is16vec8 a, b, c;
a = Is16vec8(a1, b1,c1,d1,e1,f1,g1,h1);
b = Is16vec8(a2, b2,c2,d2,e2,f2,g2,h2);
c = a * b;
cout<< I16vec8(c);
...
```

Figure 4.4 – Multiplication saturée réalisée sur des vecteurs de la classe *Is16vec8*.

Tableau 4.2 – Exemples de classes disponibles dans les librairies C++ SIMD d'Intel.

Extensions	Classe	Format des données	Type de données *	Largeur en bits	Quantité de mots	Fichier d'en-tête
MMX	I64vec1	-----	__m64	64	1	ivec.h
	Is32vec4	signé	Int	32	2	
	Iu16vec4	non signé	short	16	4	
SSE1	F32vec4	signé	float	32	4	fvec.h
SSE2	F64vec2	signé	double	64	2	dvec.h
	I128vec1	-----	__m128i	128	I	
	Iu32vec2	non-signé	Int	32	4	
	Is8vec16	signé	char	8	16	

* les types de données char, short, float et int représentent les types élémentaires du langage C.

4.2.2.3 Méthodes de programmation

Plusieurs méthodes et directives de programmation ont été rédigées afin d'assurer une bonne utilisation des interfaces. La référence [26] offre un aperçu des principales consignes à respecter. Le sujet est amplement détaillé aux références [26][7][10][11][12]. Nous ne résumerons ici que les principales. Il faut toutefois souligner que tout ce qui s'applique aux unes est, généralement, valable pour les autres puisque les classes C++ SIMD sont grandement construite d'*intrinsèques*.

Deux consignes générales ressortent du lot. La première consiste à exploiter le caractère vectoriel des *intrinsèques*. Il est évident que le parallélisme offert par la technologie SIMD est plus rentable lorsque des vecteurs de grande dimension sont manipulés. Quant à la seconde, le sujet a été légèrement abordé un peu plus haut ; lorsque les *intrinsèques* sont impliquées dans un code, la clé d'une bonne programmation est de laisser le compilateur faire son travail.

La Figure 4.5 montre une bonne façon de manipuler des vecteurs avec les instructions *intrinsèques*. Cette figure illustre le code qui implémente une multiplication de deux vecteurs. L'utilisation du « casting » de pointeur dans les instructions donne une entière liberté au compilateur afin qu'il gère efficacement l'organisation des données dans les registres. L'efficacité de l'implémentation dépend de la grandeur du vecteur. Plus le vecteur est grand et plus il sera possible d'effectuer des opérations en rafale sans avoir à vider ou recharger le pipeline du processeur. Évidemment, les performances dépendent également de la mémoire cache : un trop gros vecteur peut aussi bien entraîner un débordement de la cache.

Lorsque de grandes sections de mémoire sont sollicitées par des fonctions *intrinsèques*, il est important d'appliquer un alignement de données afin d'optimiser la performance. Par exemple, l'utilisation d'*intrinsèques* de l'extension SSE1 nécessite un alignement de données sur 16 octets. L'alignement permet, entre autres, d'organiser des sections de mémoire selon un type de données particulier. On peut ainsi traiter, par le « casting », des vecteurs de nombres réels (*floats*) en objet de type __m128. Une

instruction a spécialement été dédiée à l'alignement de données. La commande *__declspec(align(16)) float temp[400]* permet de structurer le vecteur de nombres réels *temp* en objet de type __m128.

```
#define PARALLELE 4
#define VECTEUR 400
#include <xmmintrin.h>

__declspec(align(16)) float xa[VECTEUR], xb[VECTEUR], xc[VECTEUR];
....
for (int j=0 ; j<VECTEUR ; j+=PARALLELE)
{
*(__m128 *) & xa[j] =
          _mm_mul_ps(*((__m128 *) & xa[j]), *((__m128 *) & xa[j]));
}
```

Figure 4.5 – Implémentation d'un produit scalaire avec les *intrinsèques* de l'extension SSE1

4.2.2.4 Comparaisons

Tous les mécanismes de programmation décrits ci-dessus offrent un compromis entre l'efficacité du code et la facilité d'implémentation. Généralement, l'optimisation d'un code relatif à une application donnée se fait au détriment de la facilité d'implémentation. La Figure 4.6 illustre la relation entre la facilité d'implémentation et la performance présumée pour les différentes formes de codes.

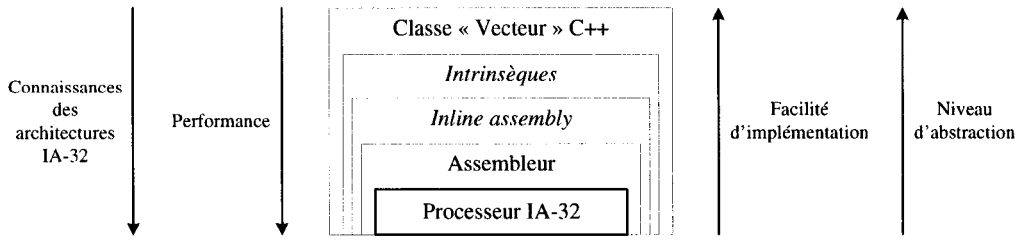

Figure 4.6 – Relation entre la performance et la facilité d'implémentation pour différents codes

Il existe plusieurs mécanismes de programmation hybride. Par exemple, on peut utiliser les types de données __M128 et les fonctions *intrinsèques* afin de passer les valeurs du C (ou C++) aux registres et routines en assembleur très optimisées qui travaillent sur ces registres. Le principe est d'utiliser les possibilités offertes par les différents mécanismes afin de créer un code optimal très rapidement. La figure suivante donne un exemple du mécanisme hybride décrit ci-dessus.

```
#include <xmmintrin.h>
….
__m128  a,b,c;
// fonctions intrinsèques
a = _mm_set_ps(4,3,2,1);
a = _mm_set_ps(8,7,6,5);

// Routine en assembleur
__asm
{
   movaps    xmm0,b
   movaps    xmm1,c
   addps     xmm0,xmm1
   movaps    a,xmm0
}

// fonctions intrinsèques
_mm_store_ps(resultants,a);
```

Figure 4.7 – Exemple de programmation assembleur par la méthode « inline assembly »

4.2.3 Bibliothèques Performance d'Intel

Les bibliothèques performance d'Intel offrent une gamme de fonctions optimisées, de très haut niveau, qui permettent d'implémenter efficacement et rapidement des fonctions relatives aux applications de traitement d'images/sons, de multimédia ou de calcul vectoriel. Ces bibliothèques ont été développées afin d'offrir la possibilité aux développeurs d'implémenter, sur un processeur d'usage générique, des applications qui

étaient jusqu'alors réservées aux DSP. Bien que ce soit entièrement transparent pour le développeur, les bibliothèques font un grand usage des extensions MMX, SSE1 et SSE2.

Le grand avantage de ces bibliothèques est qu'elles offrent une méthode très simple d'implémenter efficacement des fonctions complètes comme un filtre numérique ou une transformée de Fourier. Le principal inconvénient vient de la restriction face aux fonctions disponibles : le programmeur dispose, en effet, d'un nombre fixe de fonctions pour réaliser son application.

Parmi les différentes bibliothèques performances développées par Intel, deux ont été retenues pour l'implémentation de l'algorithme de Crozier : Signal Processing Library (SPL) et Math Kernel Library (MKL). La première contient une variété de fonctions relevant du traitement numérique de signaux : filtres numériques (FIR, IIR), transformée de Fourrier (FFT) ainsi que plusieurs opérations vectorielles. La seconde offre, quant à elle, un ensemble de différentes fonctions mathématiques dédiées à l'algèbre linaire et au calcul vectoriel.

La figure suivante illustre l'implémentation de l'opération DMA de l'algorithme de Crozier avec les fonctions de chacune des libraires mentionnées ci-dessus.

```
// SPL
#include "nsp.h"
SCplx   *S1_SPL, *S2_SPL, Z_SPL;
int n;
...
nspCopy(S1_SPL, S2_SPL, n) ;
nspConj1(S2_SPL, n);
Z_SPL = nspcDotProd(S1_SPL, S2_SPL+1, n-1);

// MKL
#include "cblas.h"
ComplexFLoat *S1_MKL, Z_MKL;
Int inc = 1 ;
...
cblas_cdotc_sub(n-1, S1_MKL, inc, S1_MKL+1, inc, &Z_MKL) ;
```

Figure 4.8 – Implémentation de la fonction DMA via les librairies SPL et MKL.

L'opération réalise un produit conjugué de vecteur de nombres complexes (équation (15)). Notez qu'avec MKL, l'opération DMA ne nécessite qu'une seule ligne de code. Ceci tient au fait qu'on peut glisser le même pointeur sur deux arguments de la fonction. Toutefois, avec la SPL on doit, d'abord, dédoubler le vecteur initial et en extraire les conjugués avant d'appliquer le produit.

4.3 PROFILAGE DE L'ALGORITHME CROZIER SUR LES PENTIUMS

L'objectif est de définir les mécanismes de programmation offrant la plus grande efficacité logicielle pour l'algorithme de Crozier. Différentes variantes architecturales et techniques logicielles ont été étudiées et comparées.

4.3.1 Mesures

Rappelons que l'algorithme de Crozier se compose des quatre fonctions : DMA, RAD, Normalisation et Pondération. Puisque les fonctions Normalisation et Pondération ne comportent aucune manipulation de vecteurs, l'optimisation de l'algorithme avec la technologie SIMD passe uniquement par les fonctions DMA et RAD. Les mesures ont donc été prises pour ces dernières ainsi que pour la fonction DMARAD qui calcule successivement les opérations DMA, Normalisation et RAD. Mis à part l'« *inline assembly* », des implémentations ont été produites pour chacune des interfaces C, *intrinsèques*, classes SIMD, MKL et SPL aux instructions IA-32, MMX, SSE1 et SSE2. Parmi celles-ci, les traitements en virgule fixe et en virgule flottante ont également été couverts. Cette dernière mesure permet de vérifier, entre autres, l'effet du compilateur. Il est à noter que l'architecture logicielle de la fonction de Normalisation dépend du type de données de l'implémentation : pour les implémentations en virgule fixe, on utilise l'algorithme de Pseudo Normalisation alors que pour celles en virgule flottante, ce sont les fonctions de la bibliothèque mathématique du langage C qui ont été utilisées. Il est important de mentionner, à ce stade, que le langage C, servant d'interface aux instructions de base des architectures IA-32, tient toujours le rôle de référence aux comparaisons et discussions sur les différents mécanismes de programmation.

Les mesures sont faites à partir d'un signal de 96 échantillons. Pour la fonction DMARAD, les mesures ont été effectuées sur 6 itérations afin d'illustrer le comportement pour le traitement complet d'un signal. Un exemple de C pour l'implémentation de l'une des fonctions est exposé en ANNEXE D. Les résultats sont tous détaillés aux Tableau 4.3 et 4.4. Le premier affiche les résultats des implémentations en nombres réels, alors que le second affiche ceux des implémentations en nombres entiers.

Tableau 4.3 – Mesures enregistrées pour les fonctions DMA, RAD et DMARAD en nombres réels

Fonction	Instructions	Interface	Cycles (requis)	Cycles (opé.)	Délai (requis) (ns)
DMA	SSE1	MKL	1025	----	1.18
	SSE1	SPL	1085	716	1.25
	IA-32	C	3800	----	4.38
	SSE1	*intrinsèque*	3900	1960	4.50
	SSE1	Class C++	69000	----	80.8
RAD	SSE1	MKL	800		0.92
	SSE1	SPL	1380	523	1.60
	IA-32	C	2100	----	2.42
	SSE1	*intrinsèque*	2460	980	2.84
DMARAD	SSE1	MKL	5900	----	6.81
	SSE1	SPL	7600	----	8.78
	SSE1	*intrinsèque*	13400	----	15.5
	IA-32	C	14200	----	16.4

Pour un bon fonctionnement, certaines implémentations ou mécanismes d'opérations nécessitent l'ajout d'opérations ou de fonctions. Ces ajouts, imputables aux mécanismes de programmation et types de données, ont un impact sur les délais. Ainsi,

les tableaux de résultats affichent deux résultats de délais. Le premier, que l'on nomme *requis*, donne le délai pour l'ensemble des opérations nécessaires au calcul d'une fonction, incluant les divers ajouts. Le second, appelé *opération*, donne le nombre de cycles pour le calcul des opérations de la fonction uniquement. Ces deux mesures permettent de comparer, opérations pour opérations, le rendement des fonctions sous diverses formes d'implémentation.

Tableau 4.4 - Mesures enregistrées pour les fonctions DMA, RAD et DMARAD en nombres entiers.

Fonction	Instructions	Interface	Format des données (bits)	Cycles (requis)	Cycles (Opé.)	Délai (requis) (ns)
DMA	SSE2	*intrinsèque*	8x16	2050	1270	1.03
	SSE1	SPL	NA	1220	900	1.4
	MMX	*intrinsèque*	4x16	3040	1220	3.51
	IA32	C	1x16	3300	----	3,81
RAD	SSE2	*intrinsèque*	8x16	2400	772	1.2
	MMX	*intrinsèque*	4x16	2180	630	2.52
	SEE1	SPL	NA	5680	2000	6.56
	IA-32	C	1x16	8160	2100	9.42
DMARAD	SSE2	*intrinsèque*	8x16	12700	----	6.35
	MMX	*intrinsèque*	4x16	11472	----	13.2
	SSE1	SPL	NA	19800	----	22.9
	IA-32	C	1x16	24800	----	28.6

Notons, finalement, que les mesures ont été effectuées sur un P3 et un P4. Le premier possède une fréquence de 866MHz, une RAM de 128 MB, une cache de 256KB et une fréquence d'horloge d'un bus de système (*system bus clock*) de 133MHz. Le P4 opère, quant à lui, à 2GHz et dispose d'une RAM de 1GB, d'un bus de système de 133

MHz et de mémoires caches de 512KB, 12KB et 8KB (L2 advanced transfer cache, micro-op trace cache et L1 data cache). Pour les Tableau 4.3 et 4.4, tous les résultats relatifs aux extensions IA-32, MMX et SSE1 proviennent du P3, alors que ceux relatifs à l'extension SSE2 ont été pris sur le P4.

4.3.2 Analyse des mesures

La première observation concerne la performance du mécanisme de programmation pour la classe C++. Bien qu'intuitivement nous puissions nous attendre à ce qu'il soit un peu plus lent que les autres mécanismes, sa piètre performance est vraiment très surprenante. Nous avons, ainsi, rapidement laissé tomber ce mécanisme de programmation.

Comme nous l'avons brièvement mentionné un plus haut, l'accélération de calcul est, dans bien des cas, limitée par l'ajout d'opérations relatives aux mécanismes d'implémentation et aux types de données. Par exemple, les fonctions *intrinsèques* (en virgule flottante), comme pour l'interface SPL (voir la Figure 4.8), requiert des opérations supplémentaires pour le dédoublement de vecteurs. En effet, pour pouvoir exécuter des opérations entres les échantillons d'un même vecteur, il faut, avant tout, copier ce même vecteur à deux endroits différents de la mémoire. Dans le cas des fonctions *intrinsèques*, cette manipulation a pour effet de faire grimper le délai de 1960 à 3900 cycles.

Le traitement en virgule fixe est un autre exemple de cas où le bon fonctionnement nécessite un ajout d'opérations. Cette fois, les opérations supplémentaires sont attribuables à la réduction d'échelle. Toutes les implémentations en virgule fixe offrent un rendement, en cycles d'horloge, inférieur à celles en virgule flottante. Toutefois, si on compare les rendements, opération pour opération, (sans réduction d'échelle) il est évident que le calcul en virgule fixe offre un potentiel d'accélération de calcul plus élevé. Dans d'autres cas, comme pour la fonction DMA implantée en virgule fixe avec les instructions SSE1 via l'interface *intrinsèque*, les deux phénomènes sont observables.

Quant aux instructions SSE2 (en virgule fixe), c'est la fréquence d'horloge du P4 qui permet de compenser l'ajout d'opérations.

En prenant en considération les délais requis, c'est l'interface MKL qui offre le meilleur rendement. Pour être plus juste, on devrait dire que l'interface MKL est celle qui répond le mieux aux spécifications de l'algorithme de Crozier. Le Tableau 4.5 montre les délais, en cycles d'horloge, accordés au traitement de l'algorithme de Crozier pour différentes largeurs d'impulsions. Nous avons comparé les performances de l'estimateur implanté avec la bibliothèque MKL à l'implémentation en C standard. Nous avons également effectué les mesures sur un processeur ARM940T cadencé à 180 MHz. Contrairement aux Pentiums, les processeurs ARM sont reconnus pour leur faible consommation de puissance. Soulignons que pour la fonction de Pondération, nous avons utilisé la représentation polaire (reformulation algorithmique) tel que décrite au CHAPITRE 3.

Tableau 4.5 - Performances (milliers de cycles) de l'estimateur en fonction du nombre d'échantillons.

Nombre d'échantillons	12	24	48	96	192	384	768	1536	3072	Pente
P3 (MKL)	2.8	3.2	5.9	8.3	12.3	19.5	33.2	59.2	119.6	38
P3 (C)	9	11.4	15.3	22.	36.5	66.2	120.6	229.4	453.5	145
ARM940T	11.8	15.9	21.6	29.4	48	82.3	151.4	290.6	562.5	180

On constate que les délais évoluent linéairement en fonction du nombre d'échantillons. Évidemment, c'est l'implémentation avec la bibliothèque MKL qui donne les meilleurs résultats. Non seulement le nombre de cycles est de loin inférieur à l'estimateur en C, mais la pente de croissance est de beaucoup inférieure. Il est évident que du point de vue du temps de calcul, le ARM940T ne se compare pas au P3. Par

contre, ce processeur a une consommation de puissance de l'ordre de 0.130W, ce qui est largement inférieur à celle du Pentium.

Le plus grand inconvénient des bibliothèques performance interfaces se situe au niveau des restrictions qu'elles imposent au concepteur. Ce dernier est restreint aux fonctions offertes par les différentes bibliothèques. Si les spécifications de l'application ne correspondent pas parfaitement à l'une des interfaces, le concepteur doit, alors, s'ajuster en ajoutant des opérations ou des fonctions supplémentaires. On retrouve, alors, l'un des cas décrits aux paragraphes précédents.

Les interfaces aux instructions IA-32, MMX et SSE1 ont également été testées sur le P4. Dans la plupart des cas, nous avons observé des mesures, en cycles d'horloge, semblables ou légèrement plus élevés que celles enregistrées sur le P3. Par exemple, la fonction DMA sous MKL passe de 1025 à 1150 cycles. La fréquence d'horloge étant beaucoup plus élevé, le délai en ns est, naturellement, réduit de 1.18 à 0.58 ns. Dans le cas du P4, nous attribuons beaucoup plus le gain en performance qui découle de son utilisation à la fréquence d'horloge qu'à son architecture.

4.4 CONCLUSION

En somme, les processeurs de la famille Pentium représentent un support matériel à une méthode de conception qui permet d'atteindre rapidement de bonnes performances. Le parallélisme et la profondeur de pipeline qu'offrent les microarchitectures et la technologie SIMD placent ces machines au rang de processeurs de haute performance. L'ensemble des commodités de programmation, mises au point par Intel, donne une variété d'interfaces, de mécanismes et d'accès à un support technique qui facilite le travail du concepteur. Nous avons, toutefois, soulevé quelques bémols à cette plate-forme et ses méthodes de conception.

La forte dissipation de puissance représente, sans aucun doute, le principal inconvénient de cette plateforme de traitement. On estime à 25-30 Watt la puissance dissipée par un P3. De plus, pour obtenir d'aussi bons résultats, il faut dédier un

processeur complet à une seule application comme celle de Crozier. Puisque Crozier est l'une des fonctions d'une application, il est facile d'imaginer qu'un nombre élevé de processeurs est nécessaire à l'implémentation d'une application complexe. On peut facilement comprendre que le coût en énergie peut grimper assez rapidement.

Un avantage relatif aux diverses interfaces d'implémentation est qu'elles permettent à un concepteur d'utiliser les facilités architecturales sans en être un expert. Il faut, toutefois, souligner que ces techniques d'implémentation sont efficaces tant et aussi longtemps que le concepteur demeure dans les sentiers battus. À la moindre incartade, le bagage de connaissances nécessaire peut vite s'accroître.

CHAPITRE 5
IMPLÉMENTATION MATÉRIELLE

Au chapitre précédent, une solution logicielle à l'implémentation de l'algorithme de Crozier a été présentée et analysée. Bien que cette solution présente plusieurs caractéristiques attrayantes, il est intéressant d'étudier les possibilités qu'offre une implémentation sur une plate-forme matérielle.

Deux raisons principales motivent l'étude et l'analyse d'une solution matérielle. D'abord, comme nous l'avons mentionné au CHAPITRE 2, l'implémentation matérielle permet d'exploiter davantage le traitement parallèle, nécessaire à l'accélération des calculs. Puisque l'efficacité de calcul est presque toujours un des grands critères de performance recherchés, une étude sur l'implémentation d'un algorithme, tel que celui de Crozier, doit normalement envisager une solution matérielle.

De plus, une solution matérielle comprend, en plus du circuit dédié ou ASIC, tout ce qui est plate-forme logicielle/matérielle, systèmes sur puces (SoC) et systèmes sur puces programmables/reconfigurables (SoP/RC). Comme nous l'avons mentionné en introduction, l'estimateur est sujet à desservir une variété d'applications. Avec leurs grandes capacités et leur flexibilité d'intégration, ces plates formes sont très intéressantes pour l'implantation d'algorithme de traitement de données. Les SoP/RC peuvent prendre la forme de plates-formes de développement telle que le ARM Integrator, dont nous discuterons un peu plus loin dans ce chapitre, ou simplement d'un FPGA muni d'un processeur embarqué.

Dans une approche matérielle, l'architecture du Crozier doit offrir un maximum de portabilité et surtout une facilité de réutilisation. On entend par « facilité de réutilisation » l'effort que doit fournir un concepteur afin de modifier, au niveau VHDL, l'architecture du module. Plus l'architecture est générique, configurable et paramétrable et plus le travail du concepteur sera facile. Il est donc très important d'investir les efforts

nécessaires afin d'adapter, dès les débuts, l'architecture en fonction de critères de réutilisation. Le défi est de rendre la structure réutilisable sans affecter la performance du module, qui reste la spécification dominante.

Ainsi, concrètement, l'objectif visé dans ce chapitre est de concevoir et réaliser un module matériel (IP) réutilisable et portable qui puisse satisfaire les critères de performance imposés par les différentes applications. Afin de rencontrer adéquatement les spécifications, il est nécessaire de mettre sur pied une bonne stratégie de conception.

5.1 DIRECTIVES DE CONCEPTION

De façon générale, les directives de conception servent de balises au concepteur afin qu'il puisse réaliser le circuit selon les spécifications établies. Les directives de conception prennent une toute autre importance lorsque la notion de réutilisation est considérée : dans ce contexte, les directives de design ne servent plus qu'au concepteur, mais à tous les concepteurs et utilisateurs qui devront utiliser, modifier ou réutiliser le même circuit. Elles permettent non seulement de garantir une facilité de réutilisation pour tous les concepteurs concernés, mais aussi d'atteindre les performances désirées en tenant compte des restrictions imposées par la nature réutilisable du circuit.

Nous avons regroupé les directives de conception selon deux catégories : Les directives pour la réutilisation et les directives pour la performance.

5.1.1 Directives pour la réutilisation

La notion de réutilisation, bien connue en informatique, est non seulement devenue un paramètre populaire de la réalisation matérielle, mais elle s'établit de plus en plus comme règle de base au design. Grâce à la rapide évolution des technologies des semi-conducteurs, la majorité des applications sont, désormais, sujettes à la réutilisation quelles que soient leurs complexités. Plusieurs grands noms de l'industrie se sont penchés sur la notion de réutilisation matérielle afin d'élaborer des méthodologies pour ce qu'on appelle désormais le « Design-Reuse ». Le document « *Méthodologie Design-*

Reuse » (MDR) développé par *Ludovic Loiseau* en collaboration avec la société *Miranda* [18] offre une bonne revue de littérature du sujet.

De façon générale, pour être réutilisable, un module HDL doit être configurable, portable, vérifiable, synthétisable et le code, lisible. Le MDR présente un ensemble de règles pour le codage et le design qui permettent au développeur d'orienter la conception selon le *design-reuse*. Toutefois, les règles présentées dans le MDR se situent principalement au niveau HDL et il nous est apparu nécessaire de définir une stratégie de réutilisation se situant à un niveau plus architecturale.

À ce niveau, la réutilisation est synonyme d'organisation : rendre une structure réutilisable consiste à organiser les fonctions du système selon un schème bien défini, afin de faciliter la configuration architecturale. Par exemple, la notion de réutilisation considérée au niveau architectural pourrait comprendre la stratégie utilisée pour la portion de contrôle ou communication du système. Ainsi, nous avons élaboré un ensemble de règles architecturales qui permettront de créer une structure configurable de l'estimateur.

5.1.1.1 Règles de Codage

L'objectif, ici, est de standardiser le plus possible les codes afin de faciliter leur déchiffrage. Il existe un ensemble de règles qui garantissent un codage clair, lisible et compréhensible. Généralement, ces règles s'appliquent à la description et à la syntaxe des codes : entête, indentation, commentaires, nom des signaux, nom des architectures, etc. Dans la mesure du possible, ces règles sont à suivre et, dans la plupart des cas, elles s'appliquent sans discernement.

Deux règles sont, à nos yeux, spécialement importantes pour faciliter le déchiffrage et la compréhension des codes VHDL. D'abord, il est absolument impératif de bien décrire et bien commenter le contenu HDL des fichiers. Il est aussi particulièrement important de définir une convention pour nommer les noms de ports d'entrées/sorties, de signaux, de variables, de procédures et autres. En plus d'assurer une certaine

homogénéité dans les codes, cette règle simplifie le processus de définition des noms. Mis à part les signaux communs, tels que l'horloge et le « reset », c'est au concepteur qu'il appartient de définir la convention qui convient le mieux au design.

L'ANNEXE E décrit brièvement la convention utilisée pour nommer les noms de ports d'entrées/sorties lors de la conception de l'IP Crozier. Un exemple de code VHDL y est également placé afin d'illustrer l'ensemble des conventions appliquées et l'allure générale du codage employé dans ce projet.

5.1.1.2 Règles de design

Les règles de design sont les techniques de conception de base qui permettent de réaliser un circuit fonctionnel et synthétisable indépendamment de la technologie et de l'application. Elles décrivent les choses à faire et à ne pas faire dans la description HDL afin d'obtenir un circuit de qualité qui soit synthétisable par la plupart des outils de synthèse. Puisque la plupart de ceux-ci travaillent au niveau RTL (Register Transfer Level), généralement, on les nomme règles de codage RTL. Le MDR décrit en détail la plupart de ces règles. Voici quelques exemples.

- Utiliser des registres indépendants de la technologie lors du codage style RTL pour la logique séquentielle.
- Utiliser des types définis de la bibliothèque IEEE et des sous-types dérivés de celle-ci.
- L'horloge et le *reset* sont des signaux globaux à l'ensemble du système; pas de logique sur l'horloge (*gated-clock*) ni sur le reset; utiliser un *reset* asynchrone.
- La description des machines à états doit se faire en au moins 2 *process* : un *process* combinatoire pour décrire le cheminement des états selon les entrées et le second, séquentiel, pour le changement effectif des états.
- Synchroniser les sorties à l'aide de registres; à partir d'une certaine complexité, il faut synchroniser également les entrées.

5.1.1.3 Règles architecturales

Les règles architecturales visent spécialement la réutilisation. Elles s'appliquent au plus haut niveau de la conception soit à l'architecture du système. L'objectif est d'organiser l'architecture afin qu'elle soit configurable, paramétrable et facilement modifiable pour l'usager qui désire intégrer, modifier ou utiliser le module. Les principales règles sont énumérées ci-dessous.

- Implémenter une architecture hautement configurable. Utiliser le plus d'éléments génériques et paramétriques possibles.
- Décrire une architecture systématisée, structurée et modulaire.
- Instancier et regrouper les portions du design dont la conception et la performance dépendent de la technologie (IP, Bibliothèques) : éléments de mémoire, module arithmétique et algorithmique. Le but visé est de simplifier le travail de l'utilisateur en rassemblant les éléments qui sont susceptibles d'être modifiés en fonction de la technologie.
- Regrouper les portions du design dont la conception et la performance doivent être indépendantes de la technologie : éléments de contrôle et de communication.

Le ou les contrôleurs constituent généralement l'élément central de l'architecture. Toute modification substantielle apportée au design a, généralement, une incidence sur les contrôleurs. Toute modification apportée aux contrôleurs a, normalement, des incidences sur diverses parties du design. Ce sont souvent des modules complexes qui méritent une attention particulière. Toujours dans un souci de minimiser le travail de l'utilisateur qui désire adapter ou modifier l'architecture de l'estimateur, il est important d'établir une stratégie spéciale pour l'implémentation de la partie contrôle du module. Compte tenu de la complexité de cette portion du design, l'objectif visé par ces règles est de pouvoir définir un ensemble de contrôleurs que l'on aura à retoucher le moins possible. Et si des retouches doivent être apportées aux contrôleurs du système, la description, l'architecture et la forme des contrôleurs doivent faciliter le travail du

programmeur. Les consignes qui suivent permettent d'établir une stratégie de contrôle qui va dans ce sens.

- Distribuer la stratégie de contrôle : utiliser plusieurs contrôleurs organisés selon un seul protocole de communication bien défini plutôt qu'un seul gros contrôleur qui gère l'ensemble du système;
- Optimiser le design final des contrôleurs et extraire, si possible, tous les éléments dépendants d'une technologie, tels que mémoire, modules mathématiques, compteurs, etc;
- Limiter les fonctions du contrôleur et si possible éviter les compteurs. Le contrôleur doit se limiter à une FSM. Notamment, il faut s'assurer que chaque module ou sous module, interagissant avec les contrôleurs, calcule son propre délai.
- Si possible, faire en sorte que tous les contrôleurs respectent le même schème d'actions et qu'ils effectuent tous la même fonction. (ex. Transfert de données);
- Autant que possible, utiliser la même description pour tous les contrôleurs : même disposition du code VHDL, même architecture (Mealy/Moore), même particularités (sorties registrées ou non).
- Implanter une similarité dans les signaux de contrôles pour tous les contrôleurs. Si possible, les contrôleurs ont tous les mêmes signaux de contrôle.

5.1.2 Directives pour la performance et l'efficacité de traitement

De façon analogue aux règles de réutilisation, il existe des trucs et techniques d'implémentation qui favorisent la performance.

La première de ces règles consiste à mettre à profit les IP et les bibliothèques offertes par la technologie visée. Les performances atteintes par ces IP sont souvent optimales et il est généralement inutile pour le concepteur de tenter de faire mieux. Pour le moment cette règle s'applique surtout à la technologie FPGA, mais éventuellement des générateurs de IP aussi riches seront disponibles sur la technologie ASIC.

La performance d'un système passe nécessairement par un pipeline profond et une grande fréquence d'horloge. Cependant, un pipeline profond n'implique pas nécessairement une grande fréquence d'horloge. Tout dépend de la nature du module et du nombre de niveaux de logique combinatoire entre les registres. De façon générale, on désire.

- Utiliser des architectures simples et très optimisées, spécialement pour les contrôleurs.
- Éviter les comparateurs et multiplexeurs impliquant de larges mots et si possible, les remplacer par des arbres de multiplexeurs et de comparateurs.
- Remplacer les compteurs par des pseudo-compteurs lorsque la présence de IP est non souhaitée. Par exemple, un contrôleur pour lequel on ne peut éviter la présence d'un compteur serait un bon endroit pour le pseudo-compteur. Plusieurs formes de pseudo-compteur existent : registre à décalage (encodage one-hot), LUT (Look Up Table), tableaux à décalages, etc.
- S'assurer que la portion du design, dont la performance est indépendante de la technologie, ait une performance suffisamment élevée pour ne pas causer de goulot d'étranglement et laisser les IP fixer la vitesse de l'horloge.
- S'assurer que les contrôleurs offrent une performance optimale et ce peu importe la technologie choisie.

À titre de guide lors de la conception, on peut considérer la fréquence d'horloge maximale pour un additionneur selon la technologie envisagée. La largeur des mots de l'additionneur dépend évidemment des besoins du design. De cette façon, on s'assure que toutes les portions du design, mises sur pied par le concepteur, auront une performance au moins supérieures à celle d'un additionneur et ainsi ne jamais être en cause dans la performance du système. Dans le cas des contrôleurs, on cherche plutôt une performance optimale afin qu'ils ne soient jamais en cause pour la performance du système. On évite, ainsi, toutes retouches apportées aux contrôleurs.

Dans les pages qui suivent il sera souvent question d'estimation de fréquence d'horloge et de surface matérielle consommée. Bien que dans le cas d'une architecture réutilisable on désire se dissocier d'une technologie précise le plus longtemps possible au cours du processus de conception, il est bien utile de pouvoir évaluer et comparer les performances de certains modules matériels. Ainsi, lorsque nécessaire, nous effectuons la synthèse d'un ou des modules à l'aide de l'outil Synplify de Synplicity. Nous considérons une technologie FPGA et plus précisément, la famille Virtex de Xilinx. Généralement, le grade de vitesse (speed-grade) est de 5. À titre de référence, des additionneurs de 10 et 32 bits fonctionnent, sur cette technologie, à des fréquences de 120 et 85 MHz.

5.2 ARCHITECTURE MATÉRIELLES DE L'« IP » CROZIER

Une première version de l'implémentation matérielle de l'estimateur a été produite préalablement à celle présentée dans les pages qui suivent. Cette première version avait pour but d'évaluer le potentiel d'accélération d'un calcul en matériel. L'architecture utilisée a été construite selon les prémices architecturales présentées au CHAPITRE 2. Aucune notion de réutilisation n'a été employée pour cette première version. Les détails de cette architecture ont fait l'objet d'un article que l'on retrouve à la référence [16].

En plus d'intégrer les principes de réutilisation décrits aux sections précédentes, la nouvelle version de l'architecture offre une très bonne efficacité de calcul. En effet, comme nous allons le constater, en utilisant un pipeline non seulement au niveau des opérations, mais aussi au niveau des impulsions, la performance de l'algorithme est grandement améliorée.

Pour en arriver à ces performances, nous avons mis sur pied une structure cellulaire qui offre une grande flexibilité architecturale.

5.2.1 Architecture cellulaire

Essentiellement, l'architecture matérielle du circuit décrit une forme régulière construite à partir **d'une même et seule** structure de base appelée cellule. L'assemblage de plusieurs cellules forme en quelque sorte un système de canalisation à l'intérieur duquel un flot de données circule tout en résolvant les équations mathématiques de l'algorithme de Crozier. La forme répétitive et modulaire qu'offre l'architecture matérielle est spécifiquement conçue pour répondre aux besoins en matière de réutilisation.

La cellule de base, illustrée à la Figure 5.1, est composée de deux éléments de mémoire (**M1** et **M2**), d'un contrôleur (**C**) et d'une unité de calculs que nous nommerons désormais opérateur (**O**).

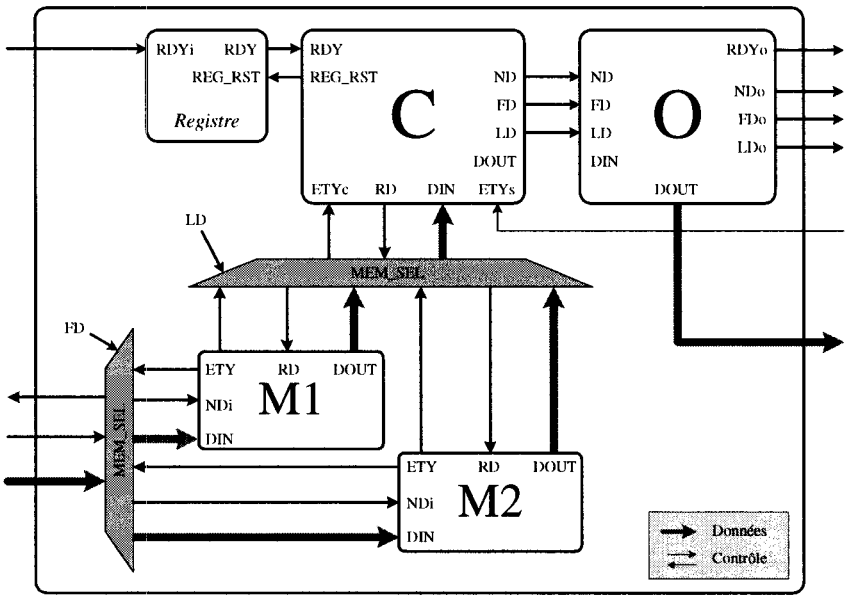

Figure 5.1 – Architecture de la cellule

L'unique rôle de la cellule est de transférer tout le contenu de l'une des mémoires à l'opérateur. Les valeurs sont alors traitées et transmises à l'une des mémoires de la

cellule subséquente et ainsi les données sont véhiculées d'une cellule à l'autre. On utilise deux éléments de mémoires pour pipeliner adéquatement les données. De cette façon, toutes les cellules travaillent simultanément. Chaque unité de calcul comporte l'une des fonctions mathématiques de l'algorithme de Crozier. L'intégration de plusieurs cellules disposées en étage, tel qu'illustré à la Figure 5.2, forme un pipeline qui propage, d'élément de mémoire en élément de mémoire, les données aux différentes fonctions mathématiques de l'estimateur. La traversée du pipeline résout l'algorithme de Crozier.

La coordination du processus de transfert de donnés entre les cellules repose sur un seul et unique protocole de communication implémenté dans les différents contrôleurs. En conséquence, les cellules ont pour seule différence majeure les fonctions mathématiques contenues dans leur opérateur. L'élaboration de l'architecture consiste donc à morceler les fonctions mathématiques de l'estimateur et à les intégrer dans les cellules.

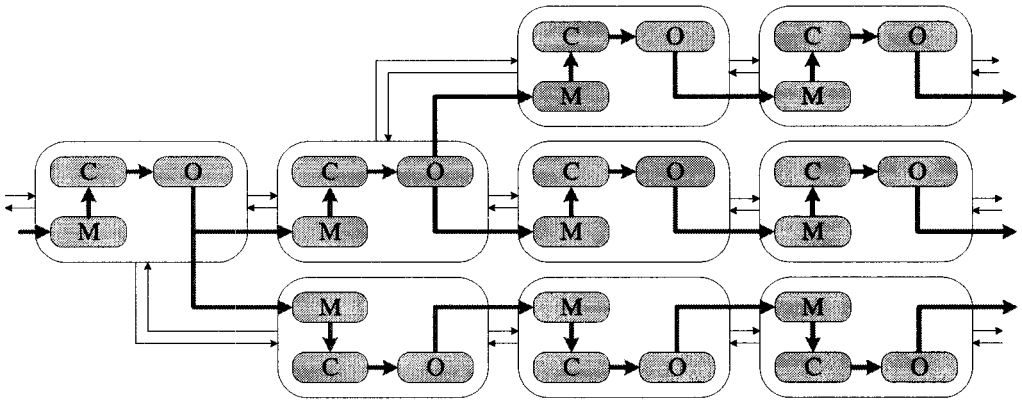

Figure 5.2 – Chemin de donnés à travers l'architecture du système

Intuitivement, on peut facilement comprendre en quoi cette architecture est très avantageuse, non seulement en matière de réutilisation, mais également au niveau de la conception. Son caractère cellulaire permet de facilement dessiner et évaluer différentes configurations. Il est aussi facile de paramétrer l'architecture en fonction d'un certain

nombre de facteurs. Par exemple, le nombre d'étages de pipeline (cellules) peut facilement être défini par une seule constante contenue dans un fichier VHDL.

Pour le concepteur qui programme en VHDL, afin de réaliser une nouvelle configuration ou modifier le module IP selon certains critères, le grand avantage de cette architecture réside dans la stratégie employée pour la portion contrôle du système. Elle répartit les fonctions de contrôle (plusieurs contrôleurs au lieu d'un seul à portée générale), elle produit une architecture uniforme et régulière des contrôleurs et finalement elle utilise un seul protocole de communication pour l'ensemble du système.

Typiquement, un design comprend un ensemble de modules locaux et un contrôleur central qui dirige et coordonne les opérations au niveau du système. Le problème avec ce type d'architecture est que chaque modification locale entraîne des répercussions sur le module central qui, à son tour, affecte les autres modules locaux. Or, pour le programmeur, ça se traduit par des modifications non seulement sur de nombreux fichiers VDHL mais, également, sur un contrôleur qui est souvent très complexe. Ce qu'offre l'architecture proposée est, d'une part, de limiter l'étendue des répercussions entraînées par les modifications locales et, d'autre part, d'offrir un ensemble de contrôleur dont l'architecture est très simple.

Il est également très important de mentionner que l'architecture présentée ci-dessus dépasse le cadre de l'estimateur Crozier. Bien qu'à l'origine l'architecture ciblait principalement ce dernier, les propriétés architecturales, élaborées afin de rendre le module configurable, nous semble, désormais, assez générique pour pouvoir servir diverses applications. Comme nous l'avons mentionné ci-dessus, la seule différence entre les cellules se trouve dans le contenu des opérateurs. Ainsi, ultimement, pour changer l'application implémentée par cette architecture, il suffit de remplacer les opérateurs par d'autres. L'architecture proposée offre donc une infrastructure et une méthodologie non seulement d'un point de vue conceptuel mais également d'un point très pragmatique, dans la mesure où un ensemble de codes sera dès le départ mis à la disposition du concepteur. Évidemment, l'application visée doit offrir un comportement

compatible à la structure présentée. *A priori*, nous visons des applications présentant un caractère algorithmique ou un flot de données doit circuler d'étage de traitement en étage de traitement. L'architecture cellulaire, présentée ici, fait l'objet d'un article que nous soumettons présentement à la conférence IWSOC 2004 (International Workshop on System-on Chip). Une copie de l'article a été mise à l'ANNEXE F.

5.2.1.1 Protocole de transfert de données

La procédure de transfert de données est réalisée à l'aide d'une machine à états dont le diagramme est illustré à la Figure 5.3. Tel que mentionné précédemment, les échantillons circulent de cellule en cellule via un pipeline de mémoires. Le travail d'une cellule donnée consiste à transférer, via un opérateur, la totalité du contenu de sa mémoire à la mémoire de la cellule subséquente. La synchronisation du transfert repose donc en grande partie sur l'état des mémoires successives et plus particulièrement l'état obtenu lorsque la mémoire est vide. C'est donc les signaux générés par l'état « vide » des mémoires successives qui fixeront les temps de départ et d'arrêt du transfert. C'est en quelque sorte un relais que les mémoires se passeront pour amorcer et arrêter leur transfert de données.

De façon séquentielle, la procédure débute avec le signal *RDY* (*ReaDY*) qui indique au contrôleur de la cellule courante la présence en mémoire d'un ensemble complet d'échantillons à traiter. Le transfert est alors amorcé et se poursuit jusqu'à ce que la mémoire soit entièrement nettoyée de son contenu, ce qui est indiqué par le signal *ETYc* (*EmpTY current*). Dès la dernière valeur lue, la mémoire courante redevient immédiatement disponible à la réception d'un autre ensemble d'échantillons. L'opérateur, quant à lui, traite les données au fur et à mesure et génère, pour chaque valeur traitée, un signal *NDo* (*New Data out*). Ce dernier est directement relié à la mémoire subséquente et lui indique la présence de données valides à inscrire. L'écriture se fait donc instantanément et sans intervention d'aucun module de contrôle. Lorsque l'opérateur a terminé sont travail, il émet le signal *RDYo* (*ReaDY out*) qui deviendra le

signal *RDY* de la cellule suivante. La cellule courante tombe alors dans un état d'attente jusqu'à la réception de l'indicateur *ETYs* (*EmpTY subsequent*).

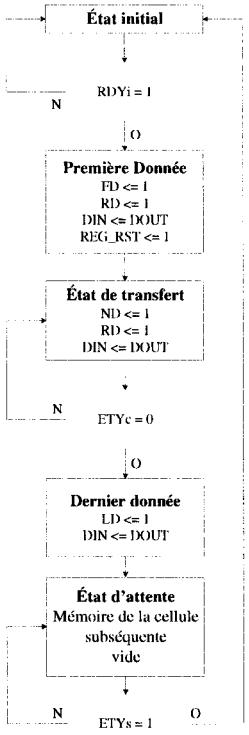

Figure 5.3 - Diagramme d'état du contrôleur

Les échantillons transférés de la mémoire à l'opérateur sont accompagnés des signaux de contrôle *FD* (*First Data*), *LD* (*Last Data*) et *ND* (*New Data*) qui assurent d'une part, la synchronisation des données et, d'autre part, le contrôle des divers sous modules de l'opérateur.

Un registre d'entré est utilisé afin de mémoriser le signal *RDYi* (*ReaDY in*) qui est susceptible de survenir à tout moment et, surtout, lorsque la cellule est en cours de transfert. Cette technique, contrairement au protocole de type *Handshaking*, permet à une cellule donnée d'émettre son signal *RDYo* sans attendre de réponse ou se soucier de la disponibilité de la cellule destinataire. En limitant ainsi les échanges de signaux, on

contribue à rendre les cellules plus indépendantes les unes des autres, ce qui représente plusieurs avantages au niveau de la conception.

Le grand avantage de cette procédure est qu'elle rend le traitement des impulsions entièrement indépendant du nombre d'échantillons du signal. La simplicité et l'efficacité des contrôleurs, qui implémentent ce protocole de transfert, sont d'autres avantages importants de cette architecture. D'une part, les contrôleurs sont entièrement dépourvus de toutes formes de compteurs et, d'autre part, leurs machines à états ne contiennent qu'un nombre assez peu élevé d'états. Ce dernier aspect se révèle très intéressant pour l'usager qui doit retoucher ou modifier les codes VHDL des contrôleurs. Rappelons que les contrôleurs représentent souvent une portion délicate d'un design. Un autre avantage relié à ce protocole est que le transfert de données peut être interrompu à tout moment. Cette propriété libère le concepteur de plusieurs considérations de synchronisation.

5.2.1.2 Cellule itérative

Parmi les algorithmes de traitement de données, on retrouve fréquemment des comportements itératifs. Il est donc très utile pour le concepteur d'avoir sous la main les commodités pour l'implémentation de composants itératifs. Avec de simples modifications, on peut facilement créer une cellule itérative à partir des bases architecturales précédemment décrites. D'abord pour la cellule, il suffit d'ajouter un élément de mémoire et un jeu de multiplexeur. La Figure 5.4 montre l'architecture de la cellule itérative.

Le troisième élément de mémoire permet à la cellule d'exécuter les itérations tout en préservant un comportement inter cellulaire adéquat : de la même manière que la cellule standard, les éléments M1 et M3 servent à véhiculer les données entre les cellules alors que l'élément M2 et l'une des mémoires M1 ou M3 servent à véhiculer les données à l'intérieur de la cellule itérative. Pour la gestion du nombre d'itérations à exécuter, un module supplémentaire (**ITE**) doit être ajouté. Son seul rôle est d'activer le signal *ITE* (*ITÉration*) lorsqu'une itération supplémentaire est à effectuer. La stratégie employée pour le calcul des itérations dépend grandement de l'application. Généralement, on peut

bénéficier des propriétés des différents opérateurs pour assurer ce travail. Les multiplexeurs sont pilotés pas les signaux de contrôle *FD*, *ND* et *LD* ainsi que les signaux *FI* (*First Iteration*), *NI* (*New Iteration*) et *LI* (*Last Iteration*) selon le fonctionnement désiré. Quant au contrôleur, il suffit d'ajouter les états première, nouvelle et dernière itération, qui permettront de délimiter le nombre d'itérations à effectuer. Le diagramme d'état du contrôleur est illustré à la Figure 5.5.

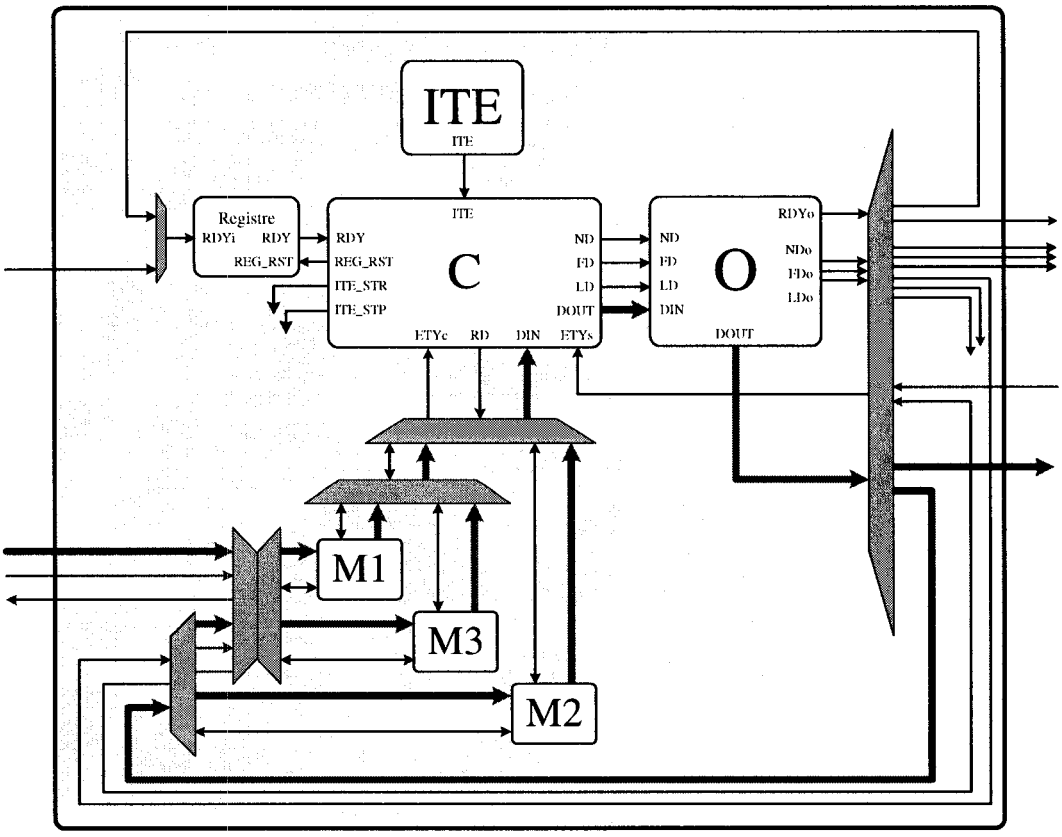

Figure 5.4 – Architecture de la cellule itérative.

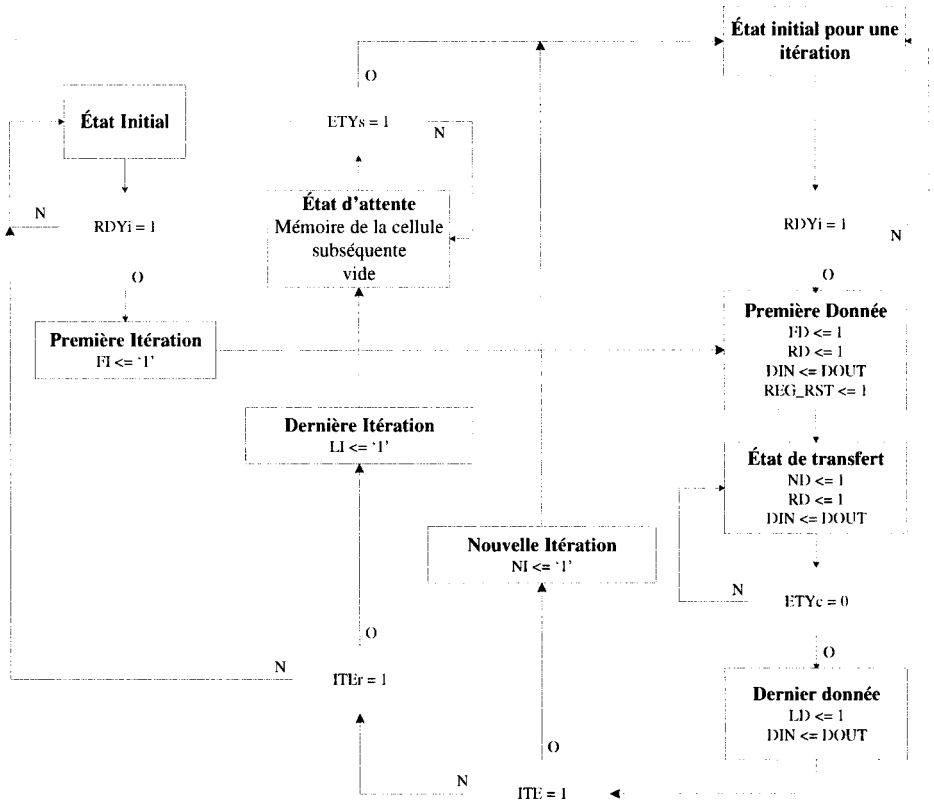

Figure 5.5 – Diagramme d'état du contrôleur de la cellule itérative.

5.2.2 Opérateurs de l'estimateur de Crozier

C'est parmi les opérateurs de l'ensemble des cellules que sont distribuées les fonctions mathématiques de l'estimateur Crozier. Rappelons que c'est uniquement par le contenu des opérateurs que les cellules se distinguent les unes des autres. Bien qu'ils représentent la portion irrégulière de l'architecture globale, les opérateurs ont, malgré tout, quelques caractéristiques communes.

- Les opérateurs sont tous composés d'un ensemble de sous opérateurs arithmétiques et d'une **ligne à délai** qui permet de distribuer vers la sortie les signaux de contrôle *LD*, *FD* et *ND*.

- L'architecture interne des sous opérateurs tels qu'additionneurs, soustracteurs, multiplieurs sont, à ce stade, volontairement indéfinis. Rappelons que selon les directives de réutilisation, l'architecture globale doit offrir un maximum d'indépendance envers les différentes technologies. Or, l'implémentation des sous opérateurs est étroitement dépendante de la technologie visée. Dans beaucoup de cas, nous pourrons bénéficier de différents IP propres à la technologie.
- Du point de vue performance, on peut s'attendre à ce que ce soit les fonctions arithmétiques ou sous opérateurs, qui limitent la fréquence d'horloge du système. Puisque *a priori* les sous opérateurs sont indéfinis, il est absolument essentiel que toute la logique agissant autour de ceux-ci fonctionne à une fréquence d'horloge supérieure à celle présumée pour ces derniers.
- Les échantillons du signal sont représentés en nombres réels de dimensions *W* bits où les parties entières et fractionnaires sont respectivement de dimensions *We* et *Wf*. Typiquement, les échantillons ont une dimension de 10 à 16 bits. Les mesures de délais et surfaces présentées ci-dessous ont toutes été faites pour le pire cas considéré, soit 16 bits.

L'estimateur Crozier compte cinq opérateurs : Pseudo Normalisateur, DMARAD, DMA, CORDIC et Pondérateur. Leurs architectures respectives sont décrites aux sections suivantes.

5.2.2.1 Pseudo Normalisateur

L'algorithme de pseudo normalisation a été décrit au CHAPITRE 3. Rappelons, rapidement, que l'objectif est de substituer la normalisation du vecteur complexe *Z* par une approximation obtenue à l'aide d'une succession de divisions par deux. De façon plus détaillée, le principe consiste à diviser par deux les composantes vectorielles, jusqu'à ce la plus grande des deux soit inférieure à 1. On obtient alors un vecteur dont la longueur est légèrement différente de la norme théorique de 1 mais qui, dans le cadre de l'estimateur, représente un résultat satisfaisant.

D'un point vu matériel, la pseudo normalisation consiste à décaler chacun des vecteurs, représentatifs des composantes vectorielles, par le bon nombre de bits. Le nombre de bits de décalage correspond à la position du '1' logique le plus significatif (pour un nombre positif, '0' pour nombre négatif) dans la portion entière du vecteur. Par exemple, pour le vecteur « 0010110,01101011 », le décalage correspondrait à 5 bits (vers la droite).

Une première solution matérielle consiste à utiliser un comparateur, un encodeur, deux décaleurs et deux multiplexeurs. En insérant un certain degré de parallélisme, il est possible de réaliser la fonction en trois cycles d'horloge.

1- Dans un premier temps, la plus grande composante vectorielle est sélectionnée par comparaison.

2- Parallèlement, tous les décalages possibles sont effectués sur les deux composantes et un encodeur détermine, à partir de la plus grande composante préalablement choisie, la position du plus significatif '1' logique (ou '0' selon le signe du mot). La position déduite correspond à la grandeur du décalage à effectuer.

3- Finalement, le résultat de l'encodage pilote les deux multiplexeurs qui sélectionnent, parmi les vecteurs préalablement décalés, ceux qui représentent le résultat de la pseudo normalisation.

L'implémentation proprement dite de cette architecture donne des estimations temporelles plutôt décevantes. Une fréquence d'horloge de moins de 60 MHz (pour des mots de 32 bits) est observée, ce qui est largement insuffisant. Le problème de cette architecture est dû à la largeur des mots que doivent traiter le comparateur, les encodeurs, décaleurs et multiplexeurs.

Pour réduire la largeur de ces mots, il suffit de dupliquer les comparateurs, encodeurs et décaleurs. De cette façon, on peut traiter parallèlement plusieurs mots de plus petite dimension plutôt qu'un seul de grande dimension. Une modification apportée

aux encodeurs et un jeu de multiplexeurs permettent de résoudre l'étape finale de sélection. La Figure 5.6 illustre l'architecture du Pseudo Normalisateur implantée dans le cadre de ce projet.

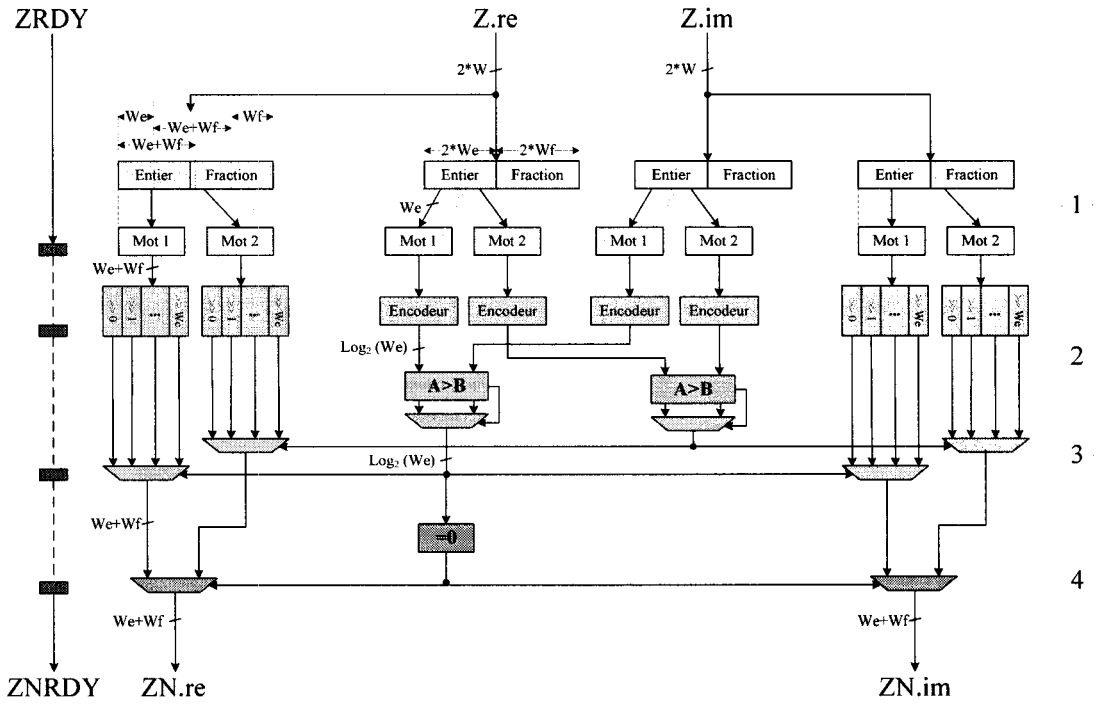

Figure 5.6 - Pseudo normalisation.

Comme on peut voir sur la figure, la première étape consiste à répartir les bits pour les composantes réelle et imaginaire du vecteur complexe. C'est à cette étape que l'on découpe les mots afin d'alléger le travail des opérateurs. Par exemple, en segmentant les parties entières de chacune des composantes en deux, on obtient quatre encodeurs qui traitent des mots de *We* bits plutôt que 2**We* bits. Les comparateurs subséquents ont des mots $\log_2(We)$ bits plutôt que $\log_2(2*We)$ bits. Un raisonnement similaire est appliqué afin de réduire la dimension des mots traités par les décaleurs et multiplexeurs. Le principe est toujours le même; il faut positionner le plus significatif '1' ou '0' logique, selon le signe du vecteur, et produire le décalage adéquat.

Les estimations donnent une fréquence d'horloge supérieure à 100 MHz (pour des mots de 32 bits) et une latence de 4 cycles d'horloge. Comparativement à la première solution proposée, le résultat est beaucoup plus satisfaisant. On peut encore accélérer le traitement en segmentant davantage les mots. Il faut toutefois prévoir une augmentation de la latence.

5.2.2.2 DMARAD et DMA

L'opérateur DMARAD, illustré à la Figure 5.7, implémente une série de modules DMA et RAD, disposés en parallèles sous deux étages : estimation et décimation. Comme nous l'avons vu (voir section 2.3.3), la concaténation des fonctions DMA et RAD en une chaîne continue d'opérations permet de créer un seul pipeline de calcul ou le traitement de l'ensemble des échantillons d'un signal donné se fait en une seule séquence. Le temps de calcul en est, conséquemment, grandement accéléré.

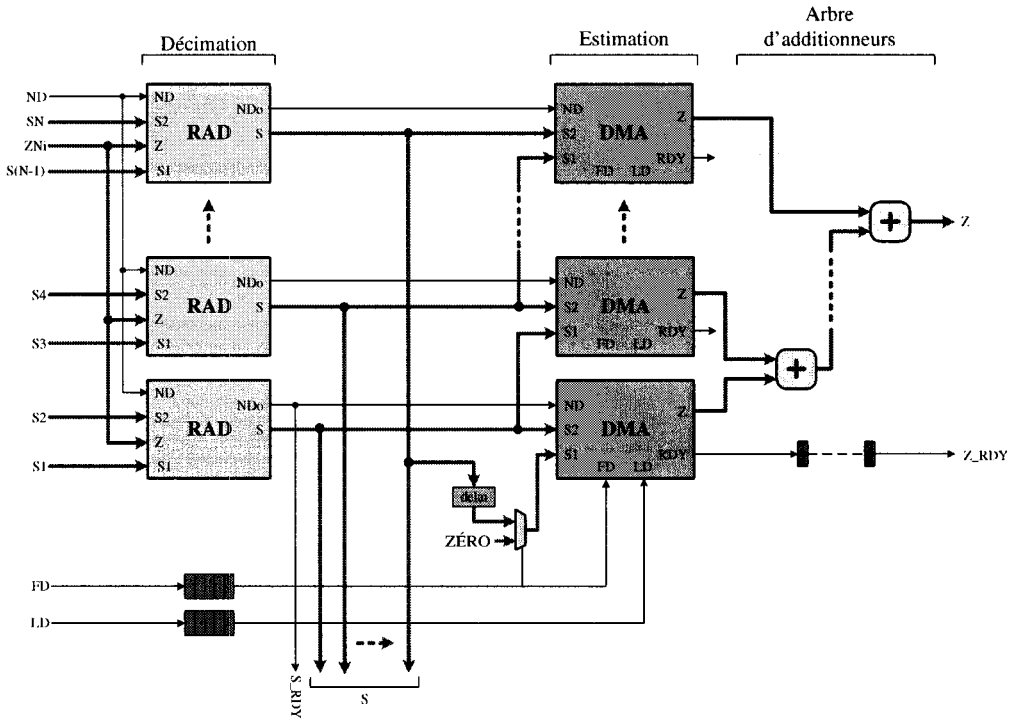

Figure 5.7 – DMARAD

Essentiellement, le comportement mathématique des fonctions RAD et DMA se résume à une multiplication de nombres complexes. Pour la fonction RAD, le résultat de la multiplication est simplement additionné à un autre échantillon du signal, alors que pour la fonction DMA, le résultat est injecté dans un accumulateur. Les architectures matérielles de ces deux fonctions sont décrites graphiquement à la Figure 5.8. Par commodité, les équations mathématiques des ces fonctions ont été retranscrites ci-dessous :

$$Z_b = \sum_{n=1}^{N-1} S_n \cdot \left(S_{n-1}\right)^* \tag{36}$$

$$S_{n,b} = S_{2n-1,b-1} + \hat{z}_{b-1}^* \cdot S_{2n,b-1} \tag{37}$$

Il est à noter que dans le cas de la fonction DMA, les accumulateurs ont été distribués et combinés aux quatre multiplicateurs d'entrée, afin de former un seul sous opérateur nommé MAC (multiplieurs-accumlateurs). Cet opérateur est très commun parmi les IP offerts par les différentes plates-formes et il est donc fort utile d'utiliser cette configuration. Pour une technologie ASIC, l'implémentation de cet opérateur est facilement réalisable à l'aide d'un multiplieur, d'un accumulateur, des multiplexeurs, des registres ainsi que des signaux *FD* et *LD* (*First Data, Last Data*).

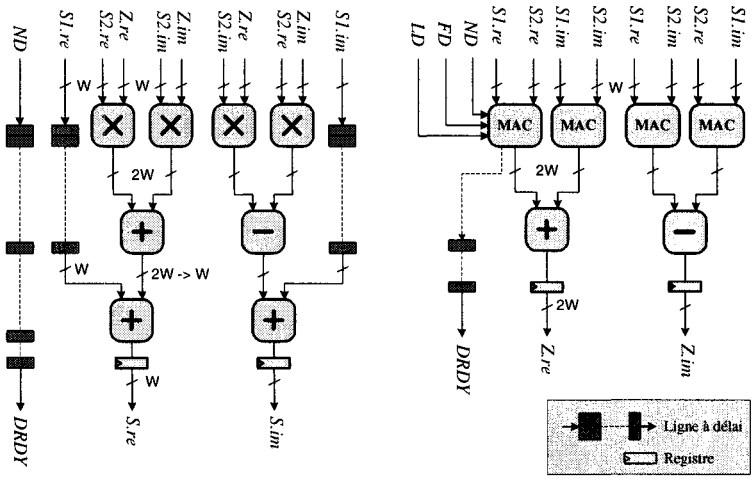

Figure 5.8 - Module RAD (à gauche) et DMA (à droite)

Revenons à l'opérateur DMARAD afin de souligner trois points importants de son architecture;

D'abord, mentionnons qu'un arbre d'additionneurs, inséré aux sorties des modules DMA, est nécessaire afin de finaliser l'estimation produite sur le signal décimé. Ce dernier complète l'accumulation des résultats générés par les sous opérateurs.

De plus, on peut immédiatement ajouter, à la suite de ce module, l'opérateur Pseudo Normalisateur dont l'architecture matérielle a été décrite à la section précédente. En insérant ce dernier à la queue de l'opérateur DMARAD, on peut former un seul module qui exécute ces deux fonctions à l'intérieur de la même cellule. Comme nous l'avons vu à la section précédente, le délai est de 4 cycles d'horloges.

Finalement, le lecteur pourra remarquer, sur la Figure 5.7, la présence d'un élément de délai et d'un multiplexeur entre les étages de décimation et d'estimation. Ces ajouts permettent de résoudre, d'une part, la multiplication entre les derniers et premiers échantillons de paquets successifs et, d'autre part, le cas ambigu du premier échantillon de chaque signal. La problématique est explicitement décrite par l'exemple suivant;

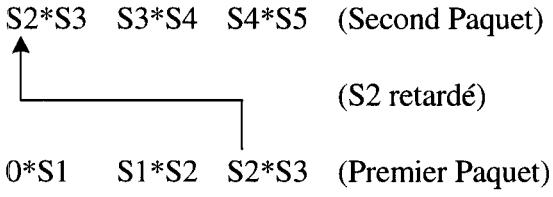

Figure 5.9 – Exemple de délai parmi les échantillons.

Pour la première vague d'échantillons, un multiplexeur piloté par le signal *FD* injecte un vecteur de zéro dans le premier opérateur DMA et ce afin de préserver l'intégrité du calcul. Par la suite, chaque échantillon sortant du troisième opérateur RAD est retardé d'un cycle d'horloge avant d'accéder à ce même opérateur DMA.

Bien qu'elle soit très avantageuse pour l'accélération de calcul, la concaténation des opérateurs RAD et DMA pose, toutefois, un léger problème. Si nous nous rapportons à la Figure 2.6, nous pouvons remarquer que, mis à part la première itération, l'algorithme exécute successivement les fonctions RAD et DMA, ce qui nous permet de les concaténer et d'obtenir, ainsi, un module de calcul très performant. Or, il reste la première itération qui, elle, n'exécute que la fonction DMA. Afin de résoudre le problème, un opérateur qui exécute seulement l'opération DMA a été créé. Son architecture est évidemment déduite de l'opérateur DMARAD. Nous verrons, plus loin dans le texte, que les cellules contenant ces opérateurs seront organisées en étages, afin de réaliser une structure configurable.

Évidemment, les caractéristiques physiques de ce module dépendent de la technologie visée et des IP utilisés pour les additionneurs et multiplieurs. À ce stade, nous sommes plus intéressés par une mesure de délai théorique qui nous permettra d'estimer les incidences de divers paramètres tels que la profondeur de pipeline des multiplieurs/additionneurs. Cependant, il faut s'assurer que la logique, environnant ces opérateurs mathématiques, est assez rapide pour ne pas faire ombrage aux performances des IP.

En se basant sur l'équation (24), on estime le délai théorique, D_D, pour les modules DMARAD et DMA, dont le degré de parallélisme est de L à

$$D_D = \left(\frac{N_b}{L} + K + 4 \right) T_{CLK} \tag{38}$$

Rappelons que K représente toujours le nombre d'étages de pipeline du module en question; T_{CLK}, la période d'horloge et que le délai accordé au Pseudo Normalisateur est de 4 cycles d'horloge.

5.2.2.3 CORDIC

La fonction du CORDIC est d'effectuer la conversion rectangulaire à polaire de l'estimation fréquentielle selon l'ensemble d'équations (34) décrit à la section 3.2.2. Son

implémentation, dont l'architecture matérielle est exposée à la Figure 5.10, est de forme itérative [2]. Principalement, l'opérateur est construit d'additionneurs/soustracteurs, de décaleurs parallèles, d'un pseudo compteur et d'une ROM. Pour chaque itération, on modifie les composantes x_i, y_i et a_i selon le signe de y_i.

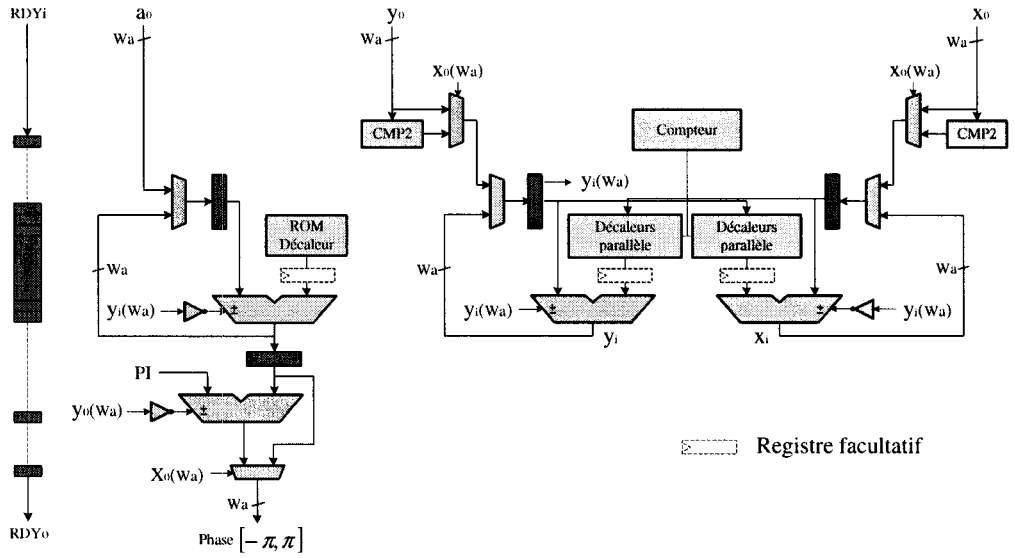

Figure 5.10 – CORDIC

La partie critique de cet algorithme se situe au niveau du décaleur de grandeur variable. Comme ce fut le cas pour le Pseudo Normalisateur, une façon efficace pour l'implémentation de cette opération consiste à produire, à chaque itération, tous les décalages possibles. On sélectionne, ensuite, le vecteur désiré à l'aide d'un multiplexeur.

Pour assumer le décompte des itérations, on utilise un pseudo compteur basé sur un type d'encodage « *one-hot* » : le compteur est remplacé par un registre à décalage. Cette solution permet d'éviter l'usage d'additionneurs dans la portion contrôle de l'opérateur. Typiquement, le nombre d'itération se situe entre 10 et 15, ce qui implique un registre de petite dimension. Cette stratégie de réalisation devrait être adaptée si le nombre d'états devient grand.

Originalement, le CORDIC est conçu pour opérer à l'intérieur de [-π/2;π/2]. On peut toutefois facilement modifier sa structure afin d'étendre sa plage d'opération à [π ; -π]. L'une des méthodes possibles consiste à effectuer une rotation supplémentaire de π selon les signes des composantes vectorielles initiales y_0 et x_0. Les relations mathématiques sont

$$\begin{aligned} x' &= d * x; \\ y' &= d * y; \\ z' &= \begin{cases} z \ si \ d = 1; \\ z + \pi \ si \ d = -1 \ et \ y_0 > 0; \\ z - \pi \ si \ d = -1 \ et \ y_0 < 0; \end{cases} \\ d &= 1 \ si \ x_0 > 0, \ -1 \ si \ x_0 < 0; \end{aligned} \tag{39}$$

Pour cette architecture le délai du CORDIC est

$$D_{CO} = (I_{CO} + 3)T_{CLK} \tag{40}$$

où I_{CO} est le nombre d'itérations utilisées pour le calcul de l'angle. Tel que défini, cet opérateur est relativement lent : en l'optimisant, on peut faire grimper sa fréquence d'horloge à un peu plus de 80 MHz, toujours pour des mots de 32 bits. Le chemin critique de l'opérateur est constitué du décaleur variable et d'un additionneur/soustracteur en série. En insérant un registre entre ces deux modules, la fréquence d'opération monte à 110 MHz, mais le nombre de cycles de latence doit, en contre partie, doubler :

$$D_{CO} = (2I_{CO} + 3)T_{CLK} \tag{41}$$

La solution restante pour accélérer le travail de l'opérateur est de réduire la largeur des mots traités.

5.2.2.4 Pondérateur

Le Pondérateur implémente l'équation (28), que nous avons retranscrit ci-dessous, ainsi que l'algorithme de contrôle nécessaire à la détection du cas de discontinuité.

$$\hat{F}_b = \hat{F}_{b-1} + (\hat{P}_b + 2\hat{P}_{b-1}) / d_b \tag{42}$$

Une solution possible pour l'architecture du Pondérateur est présentée à la Figure 5.11. L'exécution de cet opérateur se fait en 5 cycles d'horloge. La fréquence d'opération est déterminée par les opérateurs mathématiques. Puisque le Pondérateur fonctionne en tandem avec le CORDIC, il est intéressant de disposer d'une estimation du délai pour les deux opérateurs. Le délai est de

$$D_{PCO} = (I_{CO} + 8)T_{CLK} \tag{43}$$

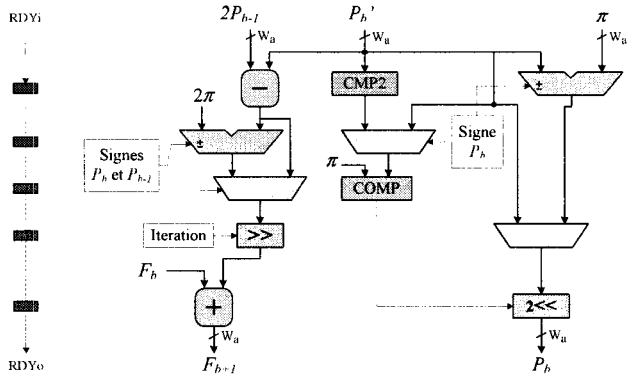

Figure 5.11 – Architecture du Pondérateur.

5.2.3 Éléments de Mémoire

Les éléments de mémoire se présentent sous deux formes différentes. La première est simplement une mémoire tampon de petite capacité. Elle est utilisée pour les cellules contenant les opérateurs CORDIC et Pondérateur. Puisque, pour ces cellules, une seule donnée est transférée à l'opérateur, le rôle de l'élément de mémoire est simplement d'absorber les délais dus à la synchronisation. Dans un contexte de réutilisation, la capacité des mémoires est un paramètre laissé, volontairement, générique afin d'économiser de l'espace.

L'autre forme d'élément de mémoire est composée d'un ensemble de mémoires d'une certaine capacité. Préférablement, des FIFOs seront utilisés. Dans le cas du Crozier, ce sont les cellules contenant les opérateurs DMA et DMARAD qui ont recours à ce type de mémoires. La capacité de ce type de mémoires doit être suffisante pour

supporter le signal complet correspondant à la cellule. De plus, l'élément de mémoire doit fournir le parallélisme inculqué aux modules DMA et DMARAD.

Selon l'architecture implémentée, le coût associé aux éléments de mémoire peut devenir considérable ; le degré de parallélisme des opérateurs, le nombre de cellules employées et la dimension des signaux traités sont tous des facteurs pouvant accroître la capacité de mémoire nécessaire. Pour le Crozier, la capacité mémoire nécessaire demeure raisonnable et ne pose pas de problème. Toutefois, pour des applications nécessitant une capacité de mémoire supérieure, il faut soit limiter au minimum le nombre de cellules ou utiliser un traitement par paquets des signaux.

5.2.4 Structure configurable

La nature cellulaire de l'architecture nous permet de créer une structure configurable en plusieurs aspects. L'une des propriétés intéressantes à paramétrer est la profondeur d'un pipeline au niveau des impulsions du signal. Rappelons qu'une impulsion est un ensemble d'échantillons correspondant au signal à traiter. Le pipeline d'impulsions représente donc la quantité de signaux à traiter qui sont simultanément présents dans l'estimateur matériel. Pour augmenter le pipeline, il suffit d'ajouter des étages de cellules contenant les opérateurs décrits ci-dessus. Généralement, la performance est tributaire d'un compromis entre la profondeur et la latence du pipeline.

Plusieurs facteurs influencent le choix d'une bonne profondeur de pipeline : la surface disponible, la largeur des impulsions, le taux de transfert et bien d'autres. L'avantage de pouvoir ajuster la profondeur du pipeline découle de la nature réutilisable de l'estimateur. Selon l'application à desservir, la profondeur de pipeline adéquate peut varier. Il est, donc, très avantageux pour le concepteur de pouvoir ajuster la profondeur de pipeline en ne modifiant qu'une constante dans un fichier VHDL.

Le principe d'une structure configurable au niveau du pipeline d'impulsion consiste à créer un estimateur itératif que l'on juxtapose à un nombre variable d'étages d'un estimateur non itératif. Le rôle de l'estimateur itératif est de compléter les itérations qui

ne seront pas traitées par la section non itérative du Crozier. Puisque le nombre d'échantillons peut varier d'une impulsion à l'autre, le Crozier itératif doit être en mesure de traiter un nombre d'itérations variable. Évidemment, le nombre d'étages non itératives ou la profondeur du pipeline d'impulsions sont fixés par de simples constantes VHDL.

Pour construire l'estimateur itératif, il suffit d'utiliser la cellules itérative, décrite à la section 5.2.1, avec les opérateurs DMARAD, CORDIC et Pondérateur. Le CORDIC, aussi utile au Crozier itératif, peut simplement être utilisé via une cellule standard. La Figure 5.12 montre l'architecture du Crozier itératif.

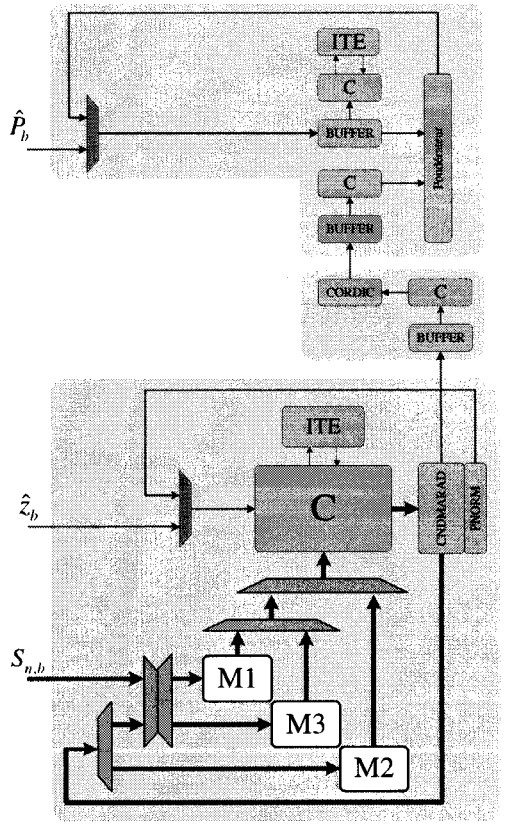

Figure 5.12 – Architecture du Crozier itératif.

Étant donné la nature réutilisable de l'estimateur, une formule d'estimation de délai est un outil essentiel au concepteur. À partir de l'équation(27), le délai pour le Crozier itératif est de

$$D_{Crozier\_ite} = \left(\sum_{b=E+1}^{B+1} \max\left[D_{PCO,b-1} \middle| D_{D,b} \right] + D_{PCO,B+1} \right) T_{CLK} \tag{44}$$

Le nombre d'itérations que doit produire le Crozier itératif dépend du nombre d'étages, E, de pipeline d'impulsions implantés (excluant l'étage itératif). Une première simplification peut, aussitôt, être appliquée à cette équation. Pour la plupart des itérations, le délai de l'opérateur DMARAD sera plus élevé que celui du Pondérateur et du CORDIC combinés. Pour les autres itérations, la différence demeure assez mince pour assumer que le délai du Pondérateur sera toujours caché derrière celui du DMARAD. Le délai de la cellule itérative DMARAD peut donc être estimé à

$$D_{D\_ite} = \left[\sum_{b=E+1}^{B+1} \frac{N_b}{L} + (B+1-E)K \right] T_{CLK} \tag{45}$$

où K représente toujours la latence de l'opérateur. Notons, toutefois, que par commodité, nous incluons dans K les 4 cycles d'horloge imputables au Pseudo Normalisateur et les quelques cycles pour le contrôleur de la cellule. Généralement un délai de 2 à 4 cycles d'horloge est nécessaire au contrôleur. Une seconde simplification peut être apportée à cette estimation en remplaçant la somme par le terme N_E/L. Ainsi, le délai pour la cellule itérative est de

$$D_{Crozier\_ite} = \left(\frac{N_E}{L} + (B+1-E)K + I_{CO} + 8 \right) T_{CLK} \tag{46}$$

Avec le Crozier itératif on peut, désormais, construire une structure configurable en ajoutant des étages composées chacune des cellules DMARAD, CORDIC et Pondérateur. Comme nous l'avons vu précédemment, le délai du module DMARAD est proportionnel au nombre d'échantillons, N_b, de l'itération courante. Puisque ce dernier diminue de moitié à chaque itération, le degré de parallélisme, L, le doit aussi afin

d'obtenir des délais semblables à chaque étages du pipeline. Puisque son délai est proportionnel à N_E (équation(46)), l'étage itératif doit avoir le même degré de parallélisme que le dernier étage de pipeline. L'architecture de la structure configurable selon le pipeline d'impulsions est illustrée à la Figure 5.13.

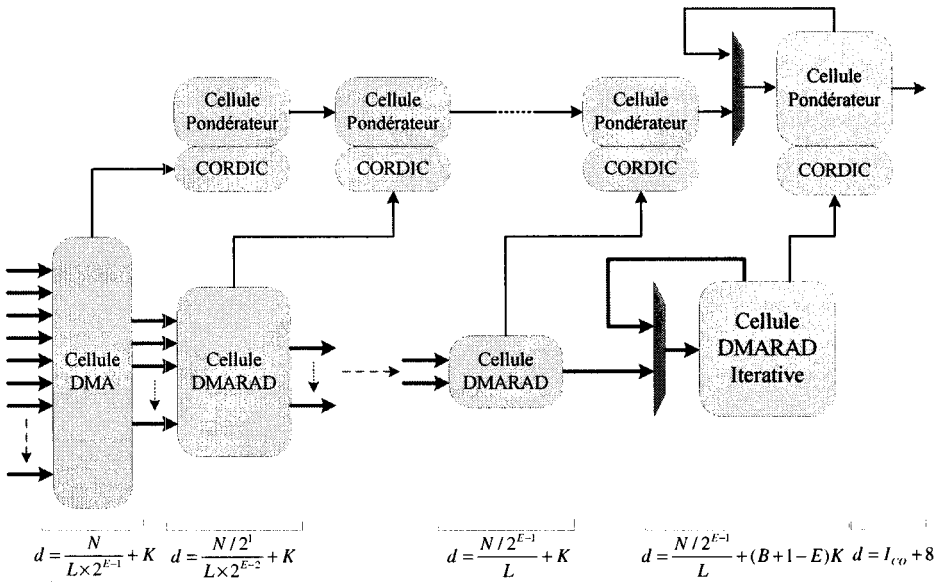

Figure 5.13 – Structure configurable de l'estimateur Crozier.

La structure fondamentale est composée uniquement d'un étage DMA et du Crozier itératif. Dans cette configuration, le niveau de parallélisme est le même pour les deux étages et correspond au niveau de base L. Selon l'équation, $B=log_2(N/3)$, qui relie les nombres d'itérations et d'échantillons, deux bases naturelles s'offrent à nous : 1 et 3.

Peu importe la largeur de l'impulsion considérée, la dernière itération aura toujours 3 échantillons à traiter. Il s'avère très utile d'avoir un degré de parallélisme, pour le module itératif, qui soit 3 fois une puissance de deux. Cette restriction permet d'éviter d'avoir à traiter, par exemple, 3 échantillons avec un module DMARAD qui possède un degré de parallélisme de 4. Évidemment, l'évolution du niveau de parallélisme au fil des étages, en débutant par l'étage itératif, se fait selon une puissance de 2, tel que $L = 2^0L$, 2^0L, 2^1L, 2^2L,... Si E représente le nombre d'étages de pipeline d'impulsions (excluant

l'étage itératif) le degré de parallélisme du Crozier est de $2^{E-1}L$. L'avantage du Crozier en base 1 est que sa configuration fondamentale offre une complexité minimale alors que l'avantage de celui en base 3 est qu'il dispose, à son stade fondamental, d'un certain niveau de parallélisme.

La latence pour le Crozier au complet est de

$$D_{Crozier} = \left[(E+1)\frac{N_E}{L} + (B+1)K + I_{CO} + 8 \right] T_{CLK} \tag{47}$$

alors que le délai par impulsion est de

$$D_{Crozier/impulsion} = \left(\frac{N_E}{L} + (B+1-E)K \right) T_{CLK} \tag{48}$$

Bien que cette équation ne soit qu'une estimation, le calcul est assez juste. Par exemple, pour un Crozier en base 1 avec E = 2, N = 96 et K = 12 nous obtenons une latence de 238 cycles et un débit de une impulsion par 96 cycles, alors que les équations nous donnent des résultats de 242 et 96. Les différences sont dues à la valeur de K qui peut varier dans le système. Par exemple, la latence des opérateurs DMA et DMARAD ne sont pas exactement les mêmes. Aussi, le nombre de cycles accordés au travail des contrôleurs peut varier de 2 à 4, ce qui entraîne, conséquemment, des fluctuations sur le résultat final. Le Tableau 5.1 montre la différence entre les mesures de délais estimés et simulés pour différentes largeurs d'impulsions.

Il faut mentionner que plus le nombre d'itérations à exécuter par l'estimateur itératif est grand et plus la latence par impulsion sera grande. Pour diminuer cette latence, on peut soit augmenter le parallélisme du module itératif, soit ajouter un autre étage de pipeline à l'entrée de ce dernier. Il est à noter la longueur minimale du signal supporté par cette architecture est de $N \geq 3x2^E$. Cette restriction nous assure que tous les étages de pipeline seront traversés par le signal.

Tableau 5.1 – Comparaison entre les mesures de délais estimés et simulés pour différentes largeurs d'impulsions (L=1, E=2).

Nombre d'échantillons		24	48	96	192	384	768	1536	3072
Estimation	Latence	102	150	234	390	690	1278	2442	4758
	Latence/imp.	36	60	96	156	264	468	864	1644
Simulation	Latence	106	158	242	398	698	1286	2446	4760
	Latence/imp.	36	62	96	158	264	468	866	1644

5.2.5 Interface et « *Wrapper* »

Pour compléter l'architecture de l'estimateur Crozier en tant que module réutilisable ou « IP », il reste à définir les caractéristiques et l'architecture de l'interface ou du « *Wrapper* » qui lui permettra de s'intégrer facilement à diverses applications.

Pour simplifier l'insertion du module à un système quelconque, l'accès à l'estimateur, que ce soit de façon directe ou par l'entremise d'une interface, se fait selon le modèle d'un FIFO : on adresse l'estimateur de la même façon que l'on adresse un FIFO. Notons que cette approche rejoint l'idée principale de l'architecture de l'estimateur, qui repose entièrement sur des éléments de mémoire. Ainsi, les signaux d'entrés de l'estimateur seront les mêmes que ceux d'une cellule : *RDY* (*ReaDY*), *FD* (*First Data*), *LD* (*Last Data*) et *ND* (*New Data*) et *ETY* (*EmpTY*).

Le principal rôle de l'interface est de relier, le cas échéant, deux domaines d'horloge différents d'un même système. Toujours en respectant une approche basée sur un élément de mémoire, on utilise un FIFO asynchrone pour faire le pont entre les domaines d'horloge. Les signaux de contrôle, qui doivent passer d'un domaine d'horloge à l'autre, sont échantillonnés afin d'éviter toute forme de métastabilité. Dans ce cas, il en a qu'un seul: *RDY*. La

Figure 5.14 montre le schéma bloc de l'interface.

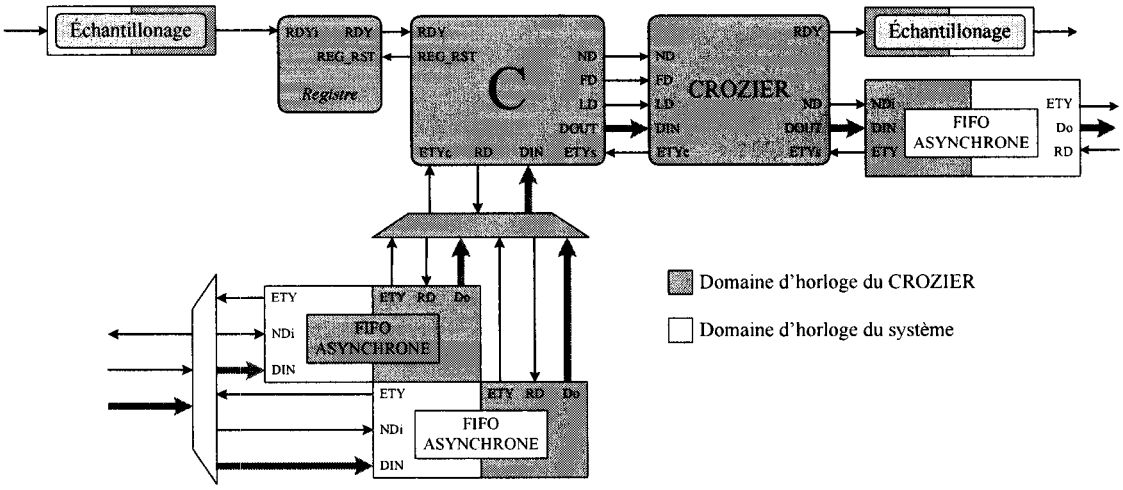

Figure 5.14 – Interface de l'estimateur.

5.3 PERFORMANCES DES OPÉRATEURS ET DE L'ESTIMATEUR

Le Tableau 5.2 montre les résultats de performance, pour divers modules, recueillis par les outils de synthèse et de placement/routage, soit Synplify de Synplicity et Design Manager de Xilinx. Les mesures de surface sont exprimées en LUT (Look Up Table) pour les résultats de synthèse et en SLICES (Unité configurable sur XCV1000) pour les résultats de placement/routage. Pour le XCV1000, une SLICE contient 2 LUTs. La pertinence d'illustrer ces deux mesures tient de l'utilisation de IP dans l'architecture. Puisque l'outil de synthèse considère les IP comme des boîtes noires, on peut, ainsi, visualiser les performances du design sans et avec certaines unités de calcul.

Il est à noter que pour les modules DMA et DMARAD, nous avons utilisé des profondeurs de pipeline de 7 et 12 (comprenant le module Pseudo Normalisateur). Soulignons, également, que les performances ont été mesurées pour le pire cas considéré, soit pour des mots d'entrés de 16 bits (W = 16) qui impliquent des

additionneurs de 32 bits. Il faut aussi mentionner que des registres d'entrée ont, parfois, été ajouté afin de bien représenter le comportement et la présence de cellules connexes qui seraient présentes en temps normal. Dans certain cas, nous avons effectué les mesures sur des Virtex de plus petite dimension, afin d'éviter la présence de longs fils qui accentuent les délais de sortie et d'entrée.

Tableau 5.2 – Performances des opérateurs.

Opérateurs	Fréquence (MHz)	LUT	SLICES (XCV1000 = 12288)	Fréquence (MHz)
	Après synthèse		Après placement&routage	
Contrôleur	192	6	4	140
Contrôleur (cellule itérative)	172	8	6	130
DMA	---	---	749	85
RAD	---	---	661	85
DMARAD (L=1)	196	160	1537	70
DMA (L=3)	120	372	2446	65
DMARAD (L=3)	110	475	3723	65
Pseudo Normalisateur	115	28	89	70
CORDIC	85	778	572	65
Pondérateur	120	128	112	75

Les résultats montrent que les contrôleurs ont tous une fréquence d'horloge supérieure à 170 MHz et ne consomment que quelques LUT (Look Up Table). La fréquence d'horloge et l'espace consommé par chaque cellule sont donc majoritairement déterminés par l'opérateur. Dans la plupart des cas, l'élément de mémoire est implanté

par un ensemble de FIFOs qui font appel aux mémoires embarquées du XCV1000. Ainsi, les performances des opérateurs représentent un bon indicateur du rendement global du système. Évidemment, les performances pour l'estimateur complet peuvent différer de celles obtenues pour un opérateur isolé.

De plus, comme pour les processeurs, les capacités de la technologie évoluent rapidement et influencent grandement les performances d'un design. Ainsi, pour que les résultats présentés au Tableau 5.2 aient un sens, il est important de les situer par rapport à d'autres FPGA. Le Tableau 5.3 montre les performances accordées à un additionneur sur trois différents FPGA de la famille Xilinx. L'additionneur a été, simplement, implanté en VHDL. Les mesures ont été prises pour des mots de 10 et 32 bits. Ces dimensions représentent les plus grands et plus petits additionneurs que l'on pourrait retrouver dans le design selon la dimension des échantillons du signal. Le tableau montre que les fréquences d'horloge peuvent varier de 30%.

Étant donné la nature reconfigurable de l'architecture, il faut aussi considérer la capacité des FPGA. Dans la famille Virtex 2, on trouve des FPGA qui offrent jusqu'à 61440 Slices (122880 LUTs). Ainsi le module DMARAD, implanté avec un degré de parallélisme de 3, (16 bits) consomme près de 30% de la surface sur le XCV1000 ou 5% de la surface d'un XC2V8000.

Tableau 5.3 – Performances de FPGA de la famille Xilinx.

Dimension (# bits)	Virtex 1 XCV50 –5 BG256		Virtex 2 XC2V250 –5 FG256		SPARTAN2 XC2S100 –5 FG256	
	Fréq (MHz)	SLICE	Fréq (MHz)	SLICE	Fréq (MHz)	SLICE
10	120	5/768	194	6/1536	140	5/1200
32	85	16/768	123	17/1536	90	17/1200

5.3.1 Comparaison logiciel/matériel

Deux approches pour l'implémentation de l'estimateur Crozier ont été abordées dans ce mémoire. Bien qu'elles ne soient pas directement en compétition, il est intéressant de comparer les performances obtenues sur des mises en œuvre logicielles et matérielles de l'estimateur. Le tableau suivant montre les résultats obtenus selon chacune des approches pour différentes largeurs d'impulsions. Les performances matérielles ont été estimées pour différentes configurations matérielles.

Tableau 5.4 – Comparaison des performances de mise en œuvre logicielles/matérielles

Nombre d'échantillons		24	48	96	192	384	768	1536	3072	Pente
Solution Logicielle P3 (866 MHz)	Latence ($x10^3$ cycles)	3.2	5.9	8.3	12.3	19.5	33.2	59.2	119.6	38
	Temps (10^{-6} s)	3.7	6.8	9.6	14.2	22.5	38.3	68.4	138.1	0.042
Solution Matérielle (50 MHz) L=1 E=2	Latence	102	150	234	390	690	1278	2442	4758	1.53
	Temps (10^{-6} s)	2.04	3	4.68	7.8	13.8	25.56	48.84	95.16	0.031
	Latence/imp.	35	60	96	156	264	468	864	1644	0.53
	Temps/imp. (10^{-6} s)	0.72	1.2	1.92	3.12	5.28	9.36	17.28	32.88	0.011
Solution Matérielle (50 MHz) L=3 E=1	Latence	82	110	154	230	370	638	1162	2198	0.69
	Temps (10^{-6} s)	1.64	2.2	3.1	4.6	7.4	12.76	23.24	43.96	0.014
	Latence/imp.	44	64	92	136	212	352	620	1144	0.36
	Temps/imp. (10^{-6} s)	0.88	1.28	1.84	2.72	4.24	7.1	12.4	22.88	0.007

Pour caractériser les performances de mise en œuvre logicielles, nous avons utilisé l'implémentation en point flottant obtenue avec la librairie MKL tel que décrite au CHAPITRE 5. Bien entendu les résultats de l'implémentation matérielle sont, en réalité,

des estimations obtenues par les équations présentées à la section 5.2.4. Le tableau montre les résultats de délais logiciels en millier de cycles, de latence et de latence par impulsion matérielle. Le tableau montre également l'équivalent en temps de ces mesures. Pour l'approche matérielle, nous avons pris une fréquence de 50 MHz, ce qui nous semblait juste considérant les performances obtenues pour chacun des modules et les performances offertes par les différentes familles de FPGA. Au compte des solutions de type matérielles, il faut aussi considérer les technologies ASIC qui pourrait peut-être offrir de meilleures fréquences d'opération. Pour les implantations matérielles, nous avons choisis deux configurations : la première est un Crozier en base 1 avec un étage de pipeline (un parallélisme d'entrée de 2 échantillons), le second est la structure fondamentale en base 3. La valeur de K est de 12 et le nombre d'itérations, $I_{CO,}$ du CORDIC est de 10.

Les résultats des mises en œuvre matérielles montrent des délais inférieurs à ceux du P3. En pipelinant au niveau des impulsions, l'implantation matérielle est de 6 à 8 fois plus rapide. Toutefois, la grande différence se situe dans la croissance du nombre de cycles en fonction du nombre d'échantillons. Le P3 offre une pente de 38, comparativement à des pentes allant de 0.36 à 1.53 pour les implantations matérielles. Avec des fréquences d'horloge aussi différentes que celles utilisées dans ce cas, cette divergence, au niveau des pentes, demeure assez discrète. Mais avec une technologie ASIC où les fréquences d'horloge peuvent atteindre plusieurs centaines de MHz, la faible croissance de la latence matérielle donnerait à l'architecture matérielle un net avantage.

Tel que mentionné en introduction de ce chapitre, l'objectif était de créer une architecture réutilisable du Crozier, tout en maintenant une performance aussi bonne que les précédentes implantations de ce dernier, notamment celle présentée à la référence [16]. Les résultats ont démontré qu'en plus de pouvoir configurer l'architecture en plusieurs points (parallélisme, pipeline) les performances étaient supérieures. Par exemple, pour un signal de 96 échantillons, la structure configurable en base 3 (L=3) nécessite 154 cycles de latence et 92 cycles de latence par impulsion, alors que celle

présentée dans [16], qui possède aussi un degré de parallélisme de 3, en demande 155 pour les deux types de latences. Mais le plus grand avantage réside dans l'indépendance de cette l'architecture face au nombre d'échantillons du signal.

5.4 ARM INTEGRATOR

Afin de valider notre architecture, nous avons implanté l'estimateur Crozier sur la plate-forme ARM Integrator qui représente un support pour le développement d'applications de type système sur puce. La plate-forme offre la possibilité d'intégrer sur un même bus de communication plusieurs modules comportant processeurs et FPGA. Le bus de communication de la plate-forme respecte la spécification AMBA et plus précisément celle des bus AHB (AMBA Advanced High-performance Bus) et APB (AMBA Advanced Peripheral Bus). Physiquement, la plate-forme, dont le schéma est illustré à la Figure 5.15, est composée d'une carte mère, de deux modules processeurs et d'un module logique.

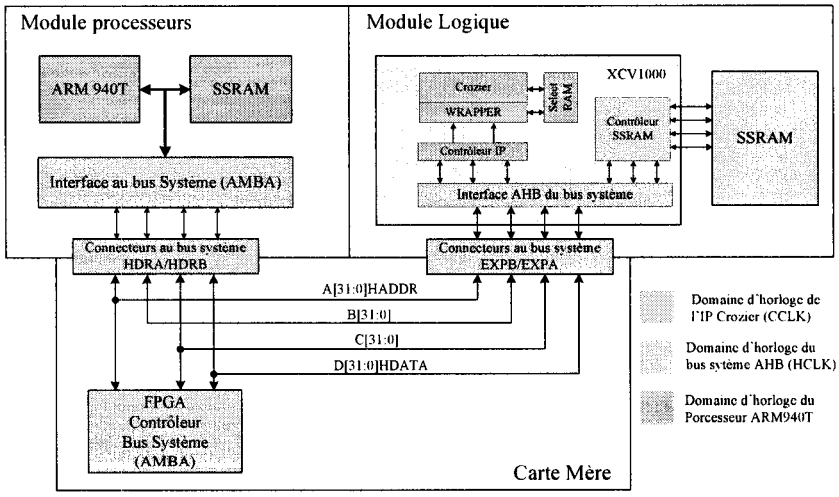

Figure 5.15 – Schéma bloc de la plate-forme ARM Integrator.

La carte mère intègre les fonctionnalités relatives à la gestion du bus de système AMBA, notamment le contrôleur, le décodeur et l'arbitre du bus. Des cartes processeurs sont disponibles avec des ARM940T et ARM7DTMI. Elles possèdent, également, des FPGA qui tiennent le rôle d'interface au bus de système. La carte logique dispose, quant à elle, d'un FPGA Xilinx XCV1000 et d'un bloc mémoire SSRAM. Il appartient à l'usager d'implanter sur le FPGA l'interface au bus. Le bus de système est composé de deux principaux bus : HADDR et HDATA, qui propagent les adresses et les données aux divers modules.

5.4.1 FPGA XCV1000

Le schéma bloc du FPGA est présenté à la Figure 5.16. En plus de l'IP Crozier, le FPGA doit supporter l'interface au bus AHB. Celle-ci comporte principalement un décodeur d'adresse qui sert à sélectionner le contrôleur du module auquel on veut accéder. Sur le schéma de la figure, deux modules sont disponibles : le Crozier et la SSRAM de la carte. Les données sont transmises via un multiplexeur. Une interface au bus périphérique complète cette portion du FPGA.

Pour l'estimateur, nous avons utilisé la structure pipelinée configurable présentée à la section 5.2.4 selon deux configurations : L=1, E=2 et L=3, E=1. Pour interfacer le Crozier au bus système, nous avons simplement remplacé les FIFO de la cellule d'entrée (DMA) par des FIFO asynchrones, tel qu'indiqué à la section 5.2.5. Les multiplieurs et additionneurs contenus dans les divers opérateurs des cellules ont tous été générés par le « coregenerator » de Xilinx. Les FIFO utilisés, sont, aussi, des IP produits par le « coregenerator ». Ils font toutefois usage de la mémoire embarquée « SelectRAM+ » du XCV1000. La dimension des mots d'entrée est de 16 bits.

Cette expérience nous a permis, d'une part, de valider le fonctionnement du IP Crozier et, d'autre part, d'évaluer la fréquence d'horloge maximale d'opération. Pour toutes les deux configurations testées, la fréquence maximale observée était de 35 MHz. Il faut toutefois mentionner qu'il est difficile de mettre à profit ce type de plate-forme afin de tester les performances d'un module matériel. La présence de processeurs, bus

système et interfaces compliquent en plusieurs points le travail de test. Par exemple, l'interface au bus système AMBA ne permet pas un taux de transfert élevé des donnés. Sous ces conditions, il est difficile d'alimenter suffisamment vite le module Crozier.

En contre partie, cette plate-forme nous a permis d'évaluer la nature réutilisable du IP Crozier et spécialement son interface. Comme nous l'avons mentionné un peu plus haut dans le texte, pour interfacer le Crozier au bus système AMBA, il a suffit de remplacer un type de FIFO par un autre.

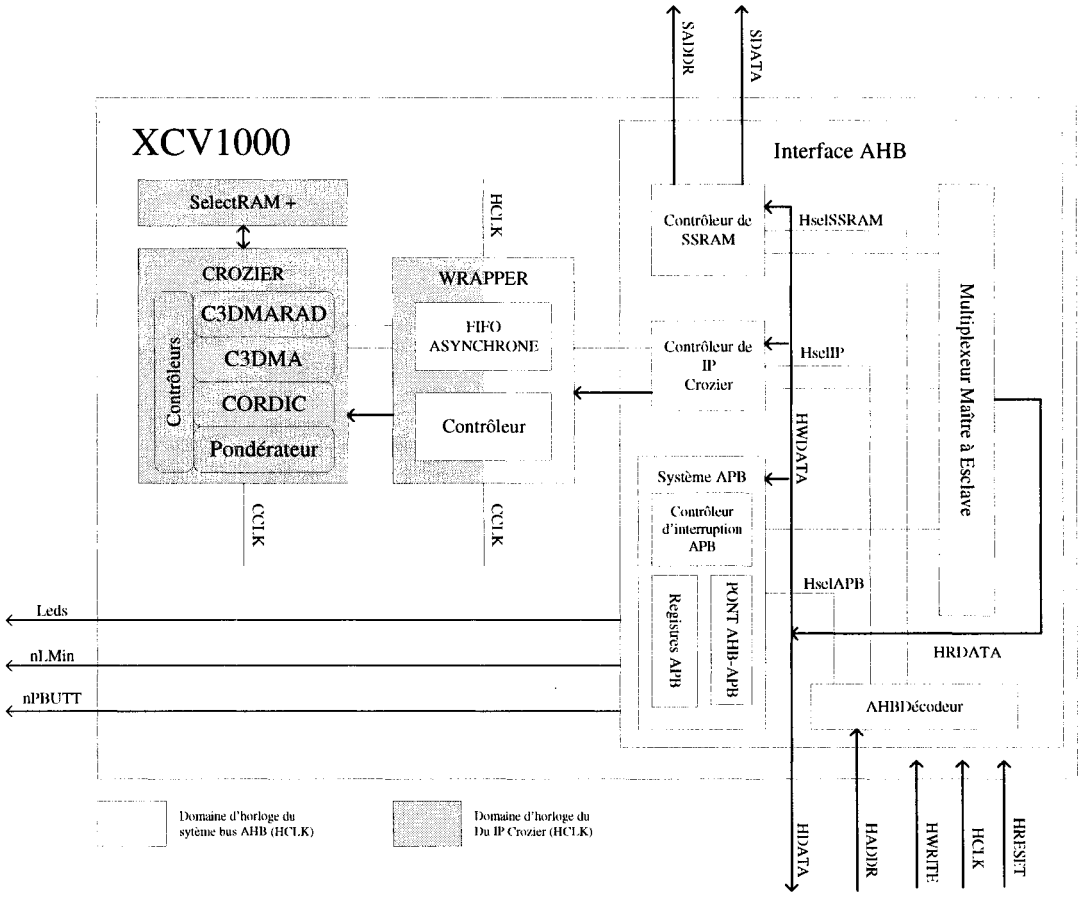

Figure 5.16 – Schéma du FPGA XCV1000.

5.5 CONCLUSION

L'étude présentée ci-dessus a démontré d'intéressants résultats de performances pour l'estimateur matériel. Mais le plus grand attrait de cette architecture réside dans son caractère réutilisable. Le concept architectural basé sur une structure cellulaire représente non seulement une bonne façon d'implémenter l'estimateur Crozier mais aussi une bonne méthodologie pour la conception de plusieurs algorithmes.

La force de cette architecture provient de la portion de contrôle. La simplicité, la régularité et la fragmentation des contrôleurs permettent au concepteur de configurer et reconfigurer facilement l'architecture. Des structures hautement pipelinées peuvent ainsi être créées afin d'offrir une intéressante puissance de calcul. La synchronisation du transfert de données autour des éléments de mémoire est un autre point fort de l'architecture. Dans le cas du Crozier, cette propriété nous a permis de mettre sur pied une architecture complètement indépendante du nombre d'échantillons contenus dans le signal.

Dans le cade de ce projet, cette architecture nous a permis d'implanter l'estimateur Crozier en une structure configurable en plusieurs points : parallélisme, nombre d'étages pipeline, nombre d'impulsions et largeur d'impulsion. Les performances enregistrées pour certaines configurations ont été plus que satisfaisantes. De plus, les propriétés de cette architecture réutilisable permettent d'interfacer facilement l'estimateur créé avec d'autres applications, ce qui est essentiel dans le domaine de la réutilisation.

CHAPITRE 6
CONCLUSION

Étant donné l'importance que prennent les estimateurs fréquentiels au sein de nombreuses applications en communication numérique, une recherche a été conduite sur l'implantation et le traitement rapide d'un algorithme efficace du nom de Crozier. Afin de répondre adéquatement aux besoins des applications, l'implantation de l'estimateur doit garantir l'efficacité de calcul et la facilité de réutilisation/intégration.

La recherche a débuté avec une analyse détaillée de l'algorithme de Crozier qui nous a permis d'établir les critères et les spécifications de la conception. Les motivations théoriques reliées à l'utilisation de l'algorithme de Crozier ont été démontrées. En plus d'avoir une complexité linéairement dépendante au nombre d'échantillons, l'estimateur de Crozier se distingue de ses compétiteurs par son seuil SNR moins élevé et une plage de fréquences effectives plus vaste. Une analyse sur la complexité de l'algorithme a démontré que ses fonctions internes sont caractérisées par un comportement antagoniste. Certaines fonctions produisent des opérations simples sur l'ensemble des échantillons du signal, alors que les autres n'ont qu'une donnée à traiter, mais dont la complexité mathématique est bien plus grande.

Pour les premières fonctions, l'accélération de calcul doit nécessairement passer par le parallélisme et le pipeline des opérations de l'algorithme. À cet effet, les prémices théoriques d'une architecture superscalaire et pipelinées ont été posées. Pour la complexité mathématique des autres fonctions, une reformulation mathématique fut nécessaire. En utilisant la représentation polaire au lieu de la représentation cartésienne pour le traitement de la phase estimée, plusieurs opérations complexes sont substituées par des simples additions ou fonctions de décalages. En contre partie, des manipulations de contrôle supplémentaires sont nécessaires afin de compenser les effets du comportement périodique relatif au travail autour du cercle polaire.

L'analyse de l'algorithme a également permis de mettre en perspective l'importance de la fonction RAD, non seulement sur l'efficacité en précision de l'estimateur, mais aussi sur l'efficacité de traitement. Bien que toute altération de cette fonction affecte l'acuité de l'estimateur, nous avons présenté des alternatives qui offrent des compromis intéressants.

Suite aux analyses, le travail d'implémentation fut amorcé. Nous avons, d'abord, mené une étude détaillée sur les performances logicielles des fonctions de l'estimateur sur les Pentiums III et IV. Les fonctions ont été implantées de plusieurs façons différentes afin de solliciter la technologie SIMD des Pentiums. Nous avons pu constater qu'en plus de donner un rendement assez surprenant, les entreprises comme Intel offrent divers mécanismes de programmation qui, bien que quelque peu contraignant, permettent aux concepteurs d'accéder aux extensions d'instructions optimisées. En contre partie, nous avons remarqué que la performance logicielle décroît rapidement en fonction du nombre d'échantillons. Il faut également souligner que la consommation de puissance est certainement l'un des désavantages marqués de ces plates-formes.

Nous avons, ensuite, étudiée une solution matérielle. Selon une approche système sur puce et dans un contexte de réutilisation, nous avons élaboré une architecture matérielle qui offre, d'une part, une grande flexibilité architecturale et, d'autre part, une très bonne efficacité de calcul. L'architecture, qui dépasse le cadre de l'estimateur Crozier, est construite à partir d'une structure cellulaire très régulière et spécialement conçue afin que l'on puisse facilement la configurer et la reconfigurer selon les spécifications désirées. En plus de permette une certaine indépendance envers la largeur des impulsions, la structure offre, entre autres, la possibilité de fixer la profondeur des pipelines. À partir de l'architecture définie, une variante a été implantée sur un XCV1000, via la plate-forme de développement ARM Integrator. Cette implantation nous a permis, d'une part, de valider l'architecture de l'estimateur et, d'autre part, d'apprivoiser une plate-forme de ce type.

En pipelinant non seulement au niveau des opérations, mais aussi au niveau des impulsions, la performance de l'estimateur matériel croît rapidement. Il faut compter une latence de l'ordre de la centaine de cycles selon la largeur des impulsions et un débit d'impulsions d'environ la moitié, selon la profondeur du pipeline. Bien qu'elle ne permet pas d'atteindre les fréquences d'horloges extrêmement élevées des processeurs génériques, l'approche matérielle, par opposition aux milliers de cycles et même aux centaines de milliers de cycles consommés par les Pentiums, demeure avantageuse.

L'architecture matérielle présentée à la dernière section de ce mémoire s'est évidemment avérée très utile pour l'estimateur Crozier, mais elle pourrait facilement s'appliquer à bien d'autres traitements. Il serait intéressant, dans de futurs travaux, de pouvoir évaluer l'utilité de cette architecture sur différentes applications. Nous estimons que pour des applications comportant un caractère algorithmique ou celles soumises à flot intensif de données, l'architecture cellulaire offre plusieurs avantages. Non seulement elle permet d'obtenir une structure configurable pour une application spécifique, mais l'infrastructure offerte, surtout en matières de codage VHDL, permet de l'obtenir rapidement. Le domaine du traitement de signal, par exemple, pourrait faire grand usage de cette architecture. Si cette architecture facilite beaucoup le travail du concepteur, il en est tout autre chose pour celui responsable de la vérification.

Évidemment, la nature configurable complique grandement le travail de vérification. Au cours de ce projet, nous nous sommes arrêtés à une vérification de type *Ad-hoc* : la vérification s'est limitée aux modules élémentaires ainsi qu'à quelques variantes de la structure configurable. Il est évident qu'une vérification beaucoup plus formelle serait nécessaire. La problématique est fort intéressante, puisqu'elle implique l'infrastructure de base, dénudée de toute application, ainsi que les architectures déduites selon les applications considérées. Tout ceci sans compter les variantes, configurations et particularités que l'on peut dériver pour chaque application. On peut facilement en déduire la complexité reliée à de tels travaux.

RÉFÉRENCES

[1] ABEYSEKERA, S. S. 1998. *"Performance of Pulse-Pair Method of Doppler Estimation"*, IEEE Trans. on Aerospace and Electronic Systems, Vol. 34, No. 2.

[2] ANDRAKA, R. 1998. *"A survey of CORDIC algorithms for FPGA based computers"*, Andraka Consulting Group, Inc, North Kingstown, Ontario, 11 pp.

[3] CANTIN, M.-A., SAVARIA, Y., LAVOIE, P. 2002. *"A Comparison of Automatic Word Length Optimization Procedures Systems"*, IEEE International Symposium on Circuits and Systems, ISCAS'2002, Vol. 2, pp. 612-615.

[4] CROZIER, S.N. 1994. *"Performance and Complexity of Discrete-Time Frequency Estimation Algorithms"*, 17th Biennial Symposium on Communications, Queen's University, Kingston, Ontario, Canada.

[5] CROZIER, S.N., MORELAND, K.W. 1992. *"Performance of a Simple Delay-Multiply-Average Technique for Frequency Estimation"*, Canadian Conference on Electrical and Computer Engineering, Toronto, Ontario, Canada, paper WM10.3, Sept. 13-26.

[6] INTEL CORPORATION. 1999. *"IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture"*. http://developer.intel.com/design/pentium4/manuals/245470.htm.

[7] INTEL CORPORATION. 1999. *"IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference"*. http://developer.intel.com/design/pentium4/manuals/245471.htm.

[8] INTEL CORPORATION, 1999. *"Intel Architecture Tutorials: Introduction to the Streaming SIMD Extensions V1.2"*.

[9] INTEL CORPORATION, 2000. *"Intel Architecture Tutorials: Introduction to the Streaming SIMD Extensions 2 for the Pentium 4 V1.3"*.

[10] INTEL CORPORATION, 2001. *"Intel C++ Compiler user's Guide and Reference"*.

[11] INTEL CORPORATION. 1999. *"Streaming SIMD Extensions Matrix Multiplication"*, App. Note AP-930, Order Number 245045-001.

[12] INTEL CORPORATION. 1999. *"Software Conventions for Streaming SIMD Extensions Version 2.1"*, App. Note AP-589, Order Number 243873-002.

[13] INTEL CORPORATION, 1999. *"Intel Math Kernel Library Reference Manual"*, Order Number 630813-008.

[14] INTEL CORPORATION, 2000. *"Intel Signal Processing1 Library Reference Manual"*, Order Number 630508-012.

[15] KAY, S. 1989. *"A fast and accurate single frequency estimator"*, IEEE Trans. Acoust., Speech, Signal Processing, Vol. 31, No 12, pp 1987-1990.

[16] LAFRANCE, L-P., CANTIN, M-A., SAVARIA, Y., SUNG, S. H., LAVOIE, P. 2002. "Architecture and Performance Characterization of Hardware and Software Implementations of the Crozier Frequency Estimation Algorithm", IEEE International Symposium on Circuits and Systems, ISCAS 2002, Vol. 4, pp 823 – 826.

[17] LANK, G. W., REED, I. S., POLLON, G. E. 1973. *"A Semicoherent Detection and Doppler Estimation Statistic,"* IEEE Trans. Aerosp. Electron. Syst., Vol. 9, No. 2, pp. 151-165.

[18] LOISEAU, L. 2001. *"Méthodologie Design Reuse"*, Miranda technologies.

[19] MANTHA, R., CROZIER, S.N. 1996. *"Implementation of a Delay-Multiply-Average frequency Estimator with Rotate-Add-Decimate Processing"*, 18th Biennial Symposium on Communications, Quenn's University, Kingston, Ontario, Canada.

[20] PATWARDHAN, B., *"Introduction to the Streaming SIMD Extensions in the Pentium III"*, National Center for Software Technology, Mumbai.

[21] RIFE, D. C. 1974. "Single-Tone Parameter Estimation from Discrete-Time Observations", IEEE Trans. on Information Theory, Vol. IT-20, No 5, pp. 591-598.

[22] SJOHOLM. S., LINDH, L. 1997. *"VHDL for Designers"*, Prentice Hall Europe.

[23] TRETTER, S.A. 1985. *"Estimating the Frequency of a Noisy Sinusoid by Linear Regression"*, IEEE Trans. on Info. Theory, Vol. IT-31, pp. 832-835.

[24] VAN TREES, H.L. 1968. "Detection, Estimation and Modulation Theory", New York, Wiley,.

[25] YOUNG, R.J., CROZIER, S.N. 1992. " *Implementation of a Simple Delay Multiply Average technique for Frequency Estimation on a Fixed Point DSP"*, Third International Symposium on Personal, Indoor and Mobile Communications, Boston, Massachusetts, paper 2.6, pp. 59-63.

[26] WOLF III, J.H. 1999. *"Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the Vtune Performance Enchancement Environnment"*, Intel Technology Journal Q2, Microprocessor Products Group, Intel Corporation.

ANNEXE A

IMPLÉMENTATION DE L'INSTRUCTION RDTSC

```
/*=============================================================================*/
/*  Very precise timers for profiling on the IA32 (Pentium or better).
/*
/* By: Francois-R Boyer (boyerf@IRO.UMontreal.CA)
/*  October 1999
/*
/*  startChrono(x)  Macro that starts the chrono x.
/*  stopChrono(x)   Macro that stops the chrono x.
/*  A chrono is a 64 bit integer global variable that counts CPU cycles.
/*  To reset a chrono you can just assing 0 to it when it is stoped.
/*  The value of a chrono is undefined while it is running so you must
/*  stop it before you access its value.
/*  The result of starting an already running chrono (or stoping an
/*  already stoped chrono) is undefined.
/*  The time spent starting and stoping chronos is not counted in the
/*  time of other running chronos.
/*
/*  Ex:
/*  __int64 a, b;
/*  ...
/*  startChrono(a);
/*  startChrono(b);
/*  stopChrono(b);
/*  stopChrono(a);
/*
/*=============================================================================*/

#include <windows.h>
#include <stdio.h>

int call_time = 0;
int start_stop_time = 0;
__int64 time_lost = 0;
// All chronos must be global variables (and should be all defined close together, to
keep them in the cache)
__int64 chrono;
__int64 c1=0, c2=0, c3=0, c4=0;

// Old Visual C doesn't have these opcodes:
#define rdtsc __asm __emit 0fh __asm __emit 031h
#define cpuid __asm __emit 0fh __asm __emit 0a2h

/*=============================================================================*/
/* Start Chrono assembly routine
/*=============================================================================*/
__declspec( naked )
void __stdcall start_chrono(__int64 *p)
{
//  rdtsc
   __asm
   {
      mov ebx, [esp+4]      // p
      push eax
      sub eax, dword ptr time_lost
      sbb edx, dword ptr time_lost+4
      sub [ebx+0], eax
      sbb [ebx+4], edx
```

```
      cpuid  // serializing instruction
      pop ecx
      sub ecx, call_time
   }

//  Must be the same in start_chrono and stop_chrono
//  (or take the same time)
   rdtsc
   __asm
   {
      sub eax,ecx
      add dword ptr time_lost, eax
      adc dword ptr time_lost+4, 0
      ret 4
   }
}

/*=============================================================================*/
/* Stop Chrono assembly routine
/*=============================================================================*/
__declspec( naked )
void __stdcall stop_chrono(__int64 *p)
{
//  rdtsc
   __asm
   {
      mov ebx, [esp+4]      // p
      push eax
      sub eax, dword ptr time_lost
      sbb edx, dword ptr time_lost+4
      sub eax, start_stop_time
      sbb edx, 0
      add [ebx+0], eax
      adc [ebx+4], edx

      cpuid  // serializing instruction
      pop ecx
      sub ecx, call_time
   }

// Must be the same in start_chrono and stop_chrono (or take the same time)
   rdtsc
   __asm
   {
      sub eax,ecx
      add dword ptr time_lost, eax
      adc dword ptr time_lost+4, 0
      ret 4
   }
}

// These macros give a little more precision (but we don't have precision under a cycle)
#define startChrono(x) { rdtsc __asm push offset x __asm call start_chrono }
#define stopChrono(x) { rdtsc __asm push offset x __asm call stop_chrono }

/*=============================================================================*/
/* Initialize Timer
/*=============================================================================*/
void init_timer()
{
// Be sure everything is in the cache
   time_lost = 0;
   call_time = 0; //start_stop_time = 0;
   startChrono(chrono);  stopChrono(chrono);

   chrono = 0;
```

```
   startChrono(chrono);
   stopChrono(chrono);
   call_time = (int)(chrono);  // This gives the time lost in the call/return and the
                               // time_lost update
   printf("call time = %d cycles\n", call_time);
}

/*=========================================================================*/
/* Main Program
/*=========================================================================*/
double difftime(time_t tim2,time_t time1);

main()
{
   double s, a, b, p1, m1, count;

   // To be even more precise, give a high priority to this process.
   SetPriorityClass(GetCurrentProcess(), 31 /*over REALTIME_PRIORITY_CLASS*/);

   // Initialize Timer
   init_timer();

   // Initialize __m64 variables
   c1 = 0; c2 = 0;

/*----------------------------------------*/
/* Measuring function atan2()
/*----------------------------------------*/

   for (j=1; j<=10000; j++)
  {
      startChrono(c1);
         s=atan2(a,b);
      stopChrono(c1);

     // Avareging the measures
      p1=double(j);
      count = (double)c1;
      m1=((p1-1)/p1)*m1 + (1/p1)*c1;
      c1=0;
   }

 // Printing the results
   printf("Delay = %lf \n", m1);

}
```

ANNEXE B

CALCUL DE RACINE POUR NOMBRES COMPLEXES

```
/*****************************************************************************
* Function Name: sqrtcplx                                                    *
* Version: 1.00                                                              *
* Author(s): P. Lavoie, A.Roussel.                                           *
* Last Modification:    23 July 1999                                         *
* (C) Copyright 1999, Her Majesty in right of Canada.                        *
* All rights reserved.                                                       *
* Translations and modifications are not permited without the consent of the *
* Defence Research Establishment Ottawa, Departement of National Defence,    *
* Ottawa, Ontario, Canada, K1A 0Z4                                           *
*----------------------------------------------------------------------------*
* History of modifications  |                                                *
*--------------------------/                                                 *
*       1.00  : Initial release                                              *
*----------------------------------------------------------------------------*
* Calling sequence  |                                                        *
-------------------/                                                         *
*       root_x = sqrtcplx(x, n);                                             *
*----------------------------------------------------------------------------*
* Inputs |                                                                   *
*-------/                                                                    *
*   <TYPE>   <var>   : DESCRIPTION                                           *
*   (ComplexDouble) x  : Complex variable                                    *
*   (unsigned) n  : exponent to perform the nth root of variable x           *
*----------------------------------------------------------------------------*
* Outputs |                                                                  *
*--------/                                                                   *
*   <TYPE>   <var>   : DESCRIPTION                                           *
*   (ComplexDouble) root_x  : nth root of x                                  *
*----------------------------------------------------------------------------*
* Returned value  |                                                          *
*----------------/                                                           *
*   Value root_x : nth root of complex variable x                            *
*----------------------------------------------------------------------------*
* Description  |                                                             *
*-------------/                                                              *
*   sqrtcplx returns x^(1/n), which is the nth root of complex variable x    *
*   note: n must be a power of 2 or 1                                        *
*----------------------------------------------------------------------------*
* Algorithm  |                                                               *
*-----------/                                                                *
*   Assume y = x / |x|                                                       *
*                                                                            *
*      if Re(y) >= 0                                                         *
*                                  1 + y                                     *
*         sqrt(y) =  -------------------------                               *
*                    sqrt(2) * sqrt(Re(1 + y))                               *
*      if Re(y) < 0                                                          *
*                   i * sign(Im(y)) * (1 - y)                                *
*         sqrt(y) = -------------------------                                *
*                   sqrt(2) * sqrt(Re(1 - y))                                *
*                                                                            *
*      sqrt(x) = sqrt(|x|) * sqrt(y)                                         *
*                                                                            *
*****************************************************************************/

ComplexDouble sqrtcplx( ComplexDouble X, unsigned ex)
{
   ComplexDouble y, ry, x;
   double M, sqt2, den;
```

```
    sqt2 = sqrt(2);
    x = X;

    if ((x.real == 0) && (x.imag == 0))
    {
       x.real = 0;
       x.imag = 0;
    }
    else if (ex >= 2)
    {
       while (ex >= 2)
       {
          M = sqrt(POW2(x.real) + POW2(x.imag));
          y.real = x.real / M;
          y.imag = x.imag / M;

          if (y.real >= 0)
          {
             den = sqt2 * sqrt(y.real + 1);
             ry.real = (y.real + 1) / den;
             ry.imag = y.imag / den;
          }
          else
          {
              if (y.imag == 0)
              {
                 den = sqt2 * sqrt(1 - y.real);
                 ry.real = y.imag / den;
                 ry.imag = (1 - y.real) / den;
              }
              else
              {
                 den = sqt2 * sqrt(1 - y.real);
                 ry.real = (SIGN(y.imag) * (y.imag)) / den;
                 ry.imag = (SIGN(y.imag) * (1 - y.real)) / den;
              }
          }

          x.real = ry.real * sqrt(M);
          x.imag = ry.imag * sqrt(M);

          ex = ex / 2;
       }
    }
    else if (ex == 1)
    {
       x = x;
    }

    return x;
}
```

ANNEXE C

ALGORITHME DE CORDIC

La résolution d'équation mathématique avec l'algorithme de CORDIC se fait selon deux modes de fonctionnement : rotation et vecteur. Les codes C ainsi qu'un tableau illustrant l'ensemble des valeurs calculées par l'algorithme son disposés ci-dessous pour chaque mode de fonctionnement.

C.1 MODE ROTATION

```
// Initialisation des variables
a = 25; // Angle initiale
x=0.607252951;
y=0;
d2=2; // Diviseur

for(i=0; i<=10; i++)
{
   d2/=2;// Multiple de 2^-i
   dx=x*d2;
   dy=y*d2;
   da=atan(d2);
   da= 180*da/PI;// Pour une valeur en degré

   if(a<0)
   {
      x += dy;
      y -= dx;
      a += da;
   }
   else
   {
      x -= dy;
      y += dx;
      a -= da;
   }
}
```

Figure C.1 - Code en langage C de l'algorithme de CORDIC en mode rotation

Tableau C.1 - Fonctionnement en mode rotation.

i	X	Y	A	dx	dy	da
0	0.607253	0.000000	25.000000	0.607253	0.000000	45.000000
1	0.607253	0.607253	-20.000000	0.303626	0.303626	26.565051
2	0.910879	0.303626	6.565051	0.227720	0.075907	14.036243
3	0.834973	0.531346	-7.471192	0.104372	0.066418	7.125016
4	0.901391	0.426975	-0.346176	0.056337	0.026686	3.576334
5	0.928077	0.370638	3.230158	0.029002	0.011582	1.789911
6	0.916495	0.399640	1.440248	0.014320	0.006244	0.895174
7	0.910250	0.413960	0.545074	0.007111	0.003234	0.447614
8	0.907016	0.421072	0.097460	0.003543	0.001645	0.223811
9	0.905371	0.424615	-0.126351	0.001768	0.000829	0.111906

C.2 MODE VECTEUR

```
// Initialisation des variables
I = 0;
x = 4;    // Coordonnée x initiale
y = 4;    // Coordonnée y initiale
a = 0;
d2 = 2; // Diviseur

for(i=0; i<15; i++)
{
   d2/= 2;     // Multiple de 2^-i
   dx=x*d2;
   dy=y*d2;
   da=atan(d2);
   da=180*da/PI; // Pour une valeur en degré

   if(y<0)
   {
      x -= dy;
      y += dx;
   a -= da;
   }
   else
   {
      x += dy;
      y -= dx;
      a += da;
   }
}
```

Figure C.2 - Code en langage C de l'algorithme de CORDIC en mode vecteur.

Tableau C.2 – Fonctionnement en mode vecteur.

I	x	y	a	dx	dy	da
0	4.000000	4.000000	0.000000	4.000000	4.000000	45.000000
1	8.000000	0.000000	45.000000	4.000000	0.000000	26.565051
2	8.000000	-4.000000	71.565051	2.000000	-1.000000	14.036243
3	9.000000	-2.000000	57.528808	1.125000	-0.250000	7.125016
4	9.250000	-0.875000	50.403791	0.578125	-0.054688	3.576334
5	9.304688	-0.296875	46.827457	0.290771	-0.009277	1.789911
6	9.313965	-0.006104	45.037546	0.145531	-0.000095	0.895174
7	9.314060	0.139427	44.142373	0.072766	0.001089	0.447614
8	9.315149	0.066661	44.589987	0.036387	0.000260	0.223811
9	9.315410	0.030274	44.813797	0.018194	0.000059	0.111906

ANNEXE D

IMPLÉMENTATION DE LA FONCTION DMARAD AVEC LES INSTRUCTIONS INTRINSÉQUES

```
/*=============================================================================
 Function     : DMARAD

 FILE         : damradsse.c
 COMMENTS     : DMARAD function implemented with SSE intrinsic instructions.
 PLATFORM     : Pentium III or higher
=============================================================================
 CREATION     : 2001/10
 AUTHOR       : Louis-Pierre Lafrance
 PROJECT      :   Implementation of a fast software Crozier function.
 LAST UPDATE : 2002/03/23
=============================================================================
 DESCRIPTION : This function calculates DMA, normisation and RAD function
               over a signal of K samples.
=============================================================================
 INPUTS
   <TYPE>   <var>    : DESCRIPTION
   (Complexfloat) *S  : Pointer to complex signal.
   (unsigned) K  : Number of samples contained in signal.
=============================================================================
 OUTPUTS
   <TYPE>   <var>    : DESCRIPTION
   (Complexfloat) *S  : Pointer to decimated complex signal.
   (Complexfloat) *Z  : Pointer to Complex frequency estimate.
=============================================================================*/

int DMARADSSE (Complexfloat *S, unsigned K, Complexfloat *Z)
{

// Memory alignment.
__declspec(align(16)) float S_re[storage_size], S_im[storage_size];
__declspec(align(16)) float S_re2[storage_size], S_im2[storage_size];
__declspec(align(16)) float S_RAD_im[storage_size], S_RAD_re[storage_size];
__declspec(align(16)) float S_RAD_im2[storage_size], S_RAD_re2[storage_size];

// 128 bits datatype
__m128 t1, t2, t3, t4, t_re, t_im, Z_re_xmm, Z_im_xmm;
float Z_im[vector_size], Z_re[vector_size];

// Other variables
Complexfloat ZZ;
float M;
int r, i, even, odd, K_2;


// Reassignment of input complex signals S. The signal has to be duplicated in order
// to perform operations.
    S[K].re =0;
    S[K].im =0;
    for (i = 0; i <= K-1 ; i++)
    {
        S_re[i]= S[i].re;
        S_im[i]= S[i].im;
        S_re2[i]= S[i+1].re;
        S_im2[i]= S[i+1].im;
    }
```

```
    // Decimating the signal in prevision of RAD function.
    K_2 = K/2;
    for (i = 0; i <= K_2-1 ; i++)
    {
        even = 2 * i;
        odd = 2 * i + 1;
        S_RAD_re[i] = S[even].re;
        S_RAD_im[i] = S[even].im;
        S_RAD_re2[i] = S[odd].re;
        S_RAD_im2[i] = S[odd].im;
    }
//-------------------------------------------------------------------------
// DMA function
//-------------------------------------------------------------------------
    for (i = 0; i <= 4 ; i++)
    {
        Z_re[i]=0;
        Z_im[i]=0;
    }

    ZZ.re = 0;
    ZZ.im = 0;

    for( i=0;i<=K-4;i+=4)
    {
        t1 = _mm_mul_ps(*((__m128 *) & S_re[i]),*((__m128 *)  &S_re2[i]));
        t2 = _mm_mul_ps(*((__m128 *) & S_im[i]),*((__m128 *)  &S_im2[i]));
        t3 = _mm_mul_ps(*((__m128 *) & S_re[i]),*((__m128 *)  &S_im2[i]));
        t4 = _mm_mul_ps(*((__m128 *) & S_im[i]),*((__m128 *)  &S_re2[i]));

        t_re = _mm_add_ps(t1,t2);
        t_im = _mm_sub_ps(t3,t4);
        *((__m128 *) &Z_im) = _mm_add_ps(*((__m128 *) &Z_im), t_im);
        *((__m128 *) &Z_re) = _mm_add_ps(*((__m128 *) &Z_re) ,t_re);
    }

    // Calculating
    ZZ.re = Z_re[0] + Z_re[1] + Z_re[2] + Z_re[3];
    ZZ.im = Z_im[0] + Z_im[1] + Z_im[2] + Z_im[3];

    // Returning the frequency estimation.
    Z->re = ZZ.re; Z->im = ZZ.im;

    //-------------------------------------------------------------------------
    // Normalization.
    //-------------------------------------------------------------------------
        M = sqrt((ZZ.re*ZZ.re)+(ZZ.im*ZZ.im));
        ZZ.re = ZZ.re/M;
        ZZ.im = ZZ.im/M;

        // Assignment of 128 bits data types.
        Z_re_xmm = _mm_set_ps1(ZZ.re);
        Z_im_xmm = _mm_set_ps1(ZZ.im);

    //-------------------------------------------------------------------------
    // RAD function
    //-------------------------------------------------------------------------
        for(i = 0; i <= K_2 -4; i+= 4)
        {
            t1 = _mm_mul_ps(Z_im_xmm,*((__m128 *)  &S_RAD_im2[i]));
            t2 = _mm_mul_ps(Z_re_xmm,*((__m128 *)  &S_RAD_re2[i]));
            t3 = _mm_mul_ps(Z_im_xmm,*((__m128 *)  &S_RAD_re2[i]));
            t4 = _mm_mul_ps(Z_re_xmm,*((__m128 *)  &S_RAD_im2[i]));

            t_re = _mm_add_ps(t1,t2);
            t_im = _mm_sub_ps(t4,t3);
```

```
            *((__m128 *) &S_re[i]) = _mm_add_ps(*((__m128 *) &S_RAD_re[i]), t_re);
            *((__m128 *) &S_im[i]) = _mm_add_ps(*((__m128 *) &S_RAD_im[i]), t_im);
        }

// Reassignment of output complex signals S..
    for (i = 0; i <= K-1 ; i++)
    {
        S[i].re=S_re[i];
        S[i].im=S_im[i];
    }

return 0;
}
```

ANNEXE E

CONVENTIONS ET CODAGE VHDL

E.1 BRÈVE DESCRIPTION DE LA CONVENTION POUR LES PORTS D'ENTRÉES/SORTIES

Pour le IP Crozier, tous les noms de ports d'entrées/sorties des entités comprises dans le module ont la forme « CXXX_YYY_ZZZ » pour lequel 'C', désigne le module IP dont l'entité fait partie ; Crozier, dans notre cas. Suivant la même logique, les ports du module Crozier ont la forme « C_ZZZ » qui a été simplifié par « CZZZ ». Le terme « XXX_YYY » est une juxtaposition de deux abréviations faisant référence à l'entité « XXX » auquel le port appartient et à l'entité « YYY », vers lequel le port se dirige ou d'où il provient. Dans l'exemple CAVCO_ND le port appartient à l'entité AVERAGE (Pondérateur) et se dirige (ou provient) de l'entité CORDIC (Pondérateur) dont l'abréviation est CO. Tous les noms de ports de l'entité AVERAGE sont de la forme « CAVYY_ZZZ » alors que ceux de l'entité CORDIC sont la forme « CCOYYY_ZZZ ». « _ZZZ » désigne la fonction du signal. Par exemple, « _ND » signifie « New Data ».

Pour plus de simplicité, la référence à l'entité de destination ou de provenance(YYY) est, dans plusieurs cas, simplement remplacée par la lettre « i » ou « o » diminutif de « in » et « out ». Par exemple, les ports CAVi_RDY CAVo_RDY. Les signaux CXi_RDY et CXo_RDY sont des signaux que l'on retrouve fréquemment dans le design.

Ainsi, selon la convention, le signal CAVCO_ND indique la présence d'une nouvelle donnée à l'entrée/sortie du module AVERAGE, provenant/allant du/au module CORDIC.

De la même façon, une norme a été définie pour les noms de signaux, variables, modules de bases et procédures. La convention devrait comporter une liste

d'abréviations clairement identifiées et commentées qui permet d'alléger la description des codes.

E.2 CODE VHDL DU MODULE « AVERAGE »

```
--=============================================================================
-- TITLE         :   AVERAGING function
-- FILE          :   CAVERAGE.vhd
-- DESCRIPTION   :   Calculation of phase averaging through iteration.
-- AUTHOR        :   Louis-P Lafrance
-- PLATFORM      :   Mixed : Independant from technology &  Xilinx Virtex
--                   Core generator IP
--=============================================================================
-- CREATION      :   2002/09
-- AUTHOR        :   Louis-P Lafrance
-- PROJECT       :   Hardware Implementation of Crozier freq. estimator IP.
-- LAST UPDATE   :   2002/11/07
--=============================================================================
-- COMMENTS      :   The averaging function calculates the average of phase
--                   estimation through iterations.
--                   - Basicaly, the function calculates
--                       Fo = Fi + (P - Pi)/Di
--                       where
--                           Fi is the frequency estimate fo the previous iteration
--                           P is current iteration estimated phase.
--                           Pi is shifted previous iteration estimated phase.
--                           Fo is the current iteration estimated frequency.
--                           Di is the delay of the given iteration.
--
--                   - However, the function prevents phase discontinuity
--                     around trigonometrical cercle by testing the phases
--                     signes before subbing them. Thus the averaging func.
--                     processes the following operations:
--                       if signe(P) /= signe(Pi)
--                           if signe(P) > 0
--                               Fo = Fi + ((P-Pi)-2pi)/Di
--                           else
--                               Fo = Fi + ((P-Pi)+2pi)/Di
--                       else
--                           Fo = Fi + (P - Pi)/Di
--                        where
--                           pi is 3.141592
--
--                   - The averaging function also provide the PHASE shifting for
--                     the next iteration according the following algorithm.
--                       if (phase) > 0) && (phase > pi/2)
--                           Po =-2*(pi-phase);
--                       else if (P) < 0) && (P < pi/2)
--                           Po =2*(pi+P);
--                       else
--                           Po = 2 *P;
--
--                   Note: CAVi_Sel indicates the shift width that
--                   corresponds to the delay to perform. A parallel shifter is
--                   used and the  CAVi_Sel signal select the corresponding shifted
--                   vector to select.
--
--                   The angles treated in this function are kept within the -pi to
--                   pi intervalle. The dimension of the words utilized are defined
--                   by the co_areg, co_areg_i and co_areg_f constants from the
--                   crozierpack package.
--
--=============================================================================
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all ;
use work.crozierpack.all;
use work.xcoregenpack.all;

entity CAVERAGE is
    generic(shifters_quty_s : integer:=0;     -- Must be different from Zero
            shifters_quty_f : integer:=crozier_p_max;
            first_iteration : integer:=1);
    port(
        CCLK      : in  std_logic;
        CRESETn   : in  std_logic;
        -- input control signals
        CAVi_Sel  : in  integer range shifters_quty_s to shifters_quty_f;
        CAVi_ND   : in  std_logic;
        CAV_P_RE_Signe : in std_logic;-- SIgne of the real component of the phase
                                      -- in rectangular representation.
        -- input DATA signals
        CAVi_P    : in  std_logic_vector(co_areg-1 downto 0);
        CAVi_Pi   : in  std_logic_vector(co_areg-1 downto 0);
        CAVi_Fi   : in  std_logic_vector(co_areg-1 downto 0);
        -- output control signals
        CAVo_WR   : out std_logic;
        CAVo_RDY  : out std_logic;
        -- output DATA signals
        CAVo_Po   : out std_logic_vector(co_areg-1 downto 0);
        CAVo_Fo   : out std_logic_vector(co_areg-1 downto 0));

end CAVERAGE;


--=============================================================================
-- Architecture definition
--=============================================================================
architecture RTL of CAVERAGE is

    component  CNSHIFTRight
    generic(cshift_N  : integer;
            cshift_w  : integer);
    port(
        CCLK        : in  std_logic;
        CRESETn     : in  std_logic;
        CSH_XIN: in  std_logic_vector(cshift_w-1 downto 0);
        CSH_XOUT: out std_logic_vector(cshift_w-1 downto 0));
    end component;

-----------------------------------------------------------------------------
-- Signal decleration
-----------------------------------------------------------------------------

    constant c_2pi : std_logic_vector(co_areg-1 downto 0):=
    "00110010010000111111011010101000";
    constant c_pi : std_logic_vector(co_areg-1 downto 0):=
    "00011001001000011111101101010100";
    constant c_pi2 : std_logic_vector(co_areg-1 downto 0):=
    "00001100100100001111110110101010";

    type lUT_XY is array (shifters_quty_s to shifters_quty_f) of
    std_logic_vector(2*sample_w-1 downto 0);
    signal lut_xout     : LUT_XY;

    -- Delay line 6 clock cycles
    signal delay_line : std_logic_vector(6 downto 0);
    signal cav_sel_t  : integer range shifters_quty_s to shifters_quty_f;
    --: std_logic_vector(log_2_x(crozier_p_max)+1 downto 0);
```

```
    signal signe_p      : std_logic;
    signal signe_pi_p   : std_logic;
    signal sel          : std_logic;
    signal p_re_signe   : std_logic;

    signal cav_p        : std_logic_vector(co_areg-1 downto 0);
    signal cav_pi       : std_logic_vector(co_areg-1 downto 0);
    signal cav_fi       : std_logic_vector(co_areg-1 downto 0);
    signal cav_fo       : std_logic_vector(co_areg-1 downto 0);
    signal cav_po       : std_logic_vector(co_areg-1 downto 0);

    signal diff_p_pi    : std_logic_vector(co_areg-1 downto 0);
    signal diff_p_pee   : std_logic_vector(co_areg-1 downto 0);
    signal diff_p_mux   : std_logic_vector(co_areg-1 downto 0);
    signal shifted_p    : std_logic_vector(co_areg-1 downto 0);
    signal diff_p_pi_2pi: std_logic_vector(co_areg-1 downto 0);
    signal po_s         : std_logic_vector(co_areg-1 downto 0);

--============================================================================
-- Architecture description
--============================================================================

begin

--============================================================================
-- Inputs process
--============================================================================
    p_inputs : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            cav_pi      <= (others=>'0');
            cav_fi      <= (others=>'0');
            cav_p       <= (others=>'0');
            cav_sel_t   <= shifters_quty_s;
            p_re_signe  <= '0';
        elsif CCLK'event and CCLK='1' then
            if CAVi_ND = '1' then
                cav_pi      <= CAVi_Pi;
                cav_fi      <= CAVi_Fi;
                cav_p       <= CAVi_P;
                cav_sel_t   <= CAVi_Sel;
                p_re_signe  <= CAV_P_RE_Signe;
            end if;
        end if;
    end process;

----------------------------------------------------------------------------
-- Signe determination for the ADDSUB module
----------------------------------------------------------------------------
    signe_pi_p <=(cav_pi(co_areg-1) xor cav_p(co_areg-1)) and p_re_signe;
    signe_p   <= cav_p(co_areg-1);


--============================================================================
-- Frequency algorithm
--============================================================================
----------------------------------------------------------------------------
-- Coregen IPs
----------------------------------------------------------------------------

    U0_F_sub_32 :  sub_32_core
    port map(
            A   => cav_p,
            B   => cav_pi,
            Q   => diff_p_pi,
            CLK => CCLK);
```

```
    -- Option for the first stage that doesn't need ajustment on the
    -- Frequency estimate.
    config_1:   if first_iteration = 0   generate
        diff_p_pi_2pi  <= diff_p_pi;
    end generate;

    config_2:   if first_iteration = 1   generate
        U1_F_addsub: addsub_32_core
        port map(
                A       => diff_p_pi,
                B       => c_2pi,
                Q_OVFL  => open,
                ADD     => signe_p,
                Q       => diff_p_pi_2pi,
                CLK     => CCLK);
    end generate;


    U2_F_add_32 :  adder_32_core
    port map(
            A   => shifted_p,
            B   => cav_fi,
            Q   => cav_fo,
            CLK => CCLK);

-------------------------------------------------------------------------------
-- Mux process: choose between the P-Pi and 2pi +or- (P-Pi).
-------------------------------------------------------------------------------
    p_mux_1 : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            diff_p_mux   <= (others=>'0');
        elsif CCLK'event and CCLK='1' then
            if signe_pi_p ='0' then
                diff_p_mux <= diff_p_pi;
            else
                diff_p_mux <= diff_p_pi_2pi;
            end if;
        end if;
    end process;

-------------------------------------------------------------------------
-- Parallel shifters generated. The outputs are stored in a LUT
--
-------------------------------------------------------------------------
    GENERATE_SHIFTERS: for index in  shifters_quty_s to shifters_quty_f generate
        US_shifter: CNSHIFTRight
            generic map(cshift_N =>index,
                        cshift_w => 2*sample_w)
                port map(
                    CCLK        => CCLK,
                    CRESETn     => CRESETn,
                    CSH_XIN     => diff_p_mux,
                    CSH_XOUT    => lut_xout(index));
    end generate;

-------------------------------------------------------------------------------
-- Selection of the good shifted signal.
-------------------------------------------------------------------------------
    p_select_shifted : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            shifted_p <= (others=>'0');
        elsif CCLK'event and CCLK='1' then
             shifted_p <= lut_xout(cav_sel_t);
        end if;
```

```
    end process;

--=============================================================================
-- Phase shifting algorithm
--=============================================================================
-----------------------------------------------------------------------------
--  Coregen IPs
-----------------------------------------------------------------------------

    U3_P_addsub: addsub_32_core
    port map(
        A=> cav_p,
        B=> c_pi,
        Q_OVFL=> open,
        ADD=> signe_p,
        Q=> diff_p_pee,
        CLK=> CCLK);


-------------------------------------------------------------------------------------------
-- Mux process: choose between 2'scomplement of P and P.
-------------------------------------------------------------------------------------------
    p_mux_2 : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            po_s      <= (others=>'0');
        elsif CCLK'event and CCLK='1' then
            if p_re_signe ='0' then
                po_s <= cav_p;
            else
                po_s <= diff_p_pee;
            end if;
        end if;
    end process;

-------------------------------------------------------------------------------------------
-- Shift process for the multiplcation by 2
--***
--  a '0' is always added.
--***
-------------------------------------------------------------------------------------------
    p_shifter : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            cav_po   <= (others=>'0');
        elsif CCLK'event and CCLK='1' then
            cav_po  <= po_s(co_areg-2 downto 0) & '0';
        end if;
    end process;
-------------------------------------------------------------------------------------------
-- Delay Line
-------------------------------------------------------------------------------------------
    p_delay_line : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            delay_line   <= (others=>'0');
        elsif CCLK'event and CCLK='1' then
            if CAVi_ND = '1' then
                delay_line   <= '1' & "000000";
            else
                delay_line   <= '0' & delay_line(6 downto 1);
            end if;
        end if;
    end process;

--=============================================================================
-- Outputs process
```

```
--=============================================================================
    p_outputs : process(CRESETn, CCLK)
    begin
        if CRESETn = '0' then
            CAVo_Po  <= (others=>'0');
            CAVo_Fo  <= (others=>'0');
            CAVo_WR  <= '0';
            CAVo_RDY <= '0';
        elsif CCLK'event and CCLK='1' then
            CAVo_Po   <= cav_po;
            CAVo_Fo  <= cav_fo;
            CAVo_RDY <= delay_line(0);
            CAVo_WR  <= delay_line(1);
        end if;
    end process;


--=============================================================================
end RTL;
--=============================================================================
```

ANNEXE F

A FRAMEWORK FOR IMPLEMENTING REUSABLE DIGITAL SIGNAL PROCESSING MODULES

L-P. Lafrance[1], Y. Savaria[1]

[1] Electrical and Computer Engineering Department, École Polytechnique de Montréal, P.O. Box 6079, Station Centre-ville, Montréal, Québec, Canada, H3C 3A7

Abstract

A framework for implementing reusable digital signal processing modules is presented. Based on a cellular structure, it offers a high level of configurability. With its predefined control strategy, and its generic architecture, the framework allows a fast and efficient implementation of digital signal processing applications. As a practical example, the implementation of a frequency estimator, called the Crozier algorithm, is presented. Advantages observed when implementing the algorithm with the proposed framework are demonstrated.

1. Introduction

With the growing complexity of Integrated Circuits (ICs), using reuse strategies for hardware design is a necessity. Nowadays, thousands of designers are creating intellectual property (IP) modules on a large scale, targeting very popular SoC (System on Chip) and SoRC (System on Reprogrammable Chip). Although reuse of hardware modules is becoming common to system design, we found that there is very little reuse between IP modules. The typical reusable component is an application specific configurable module (ASCM) and its reuse capabilities are restrained to the scope of its application.

Design for Reuse methodologies were proposed 0[2][3]. Most of these methods are HDLs (Hardware Description Languages) coding and synthesis guidelines to facilitate reuse. For instance, using generic parameters allows configurability, which is crucial for reuse. Although many rules are stated for HDL coding, there is no architecture

specification at the RT (Register Transfer) level. Therefore, no common architectural characteristics exist between the modules created under these methodologies, and the resulting components are usually ASCMs. Beside reuse methodologies, some reusable RTL architectures were proposed. But again, they target specific applications. Highly parameterized Digital Signal Processing (DSP) filters are often implemented as ASCM. References [4]0 and [5] are examples of it. The problem remains the same: the components may be reusable but their architectures are not.

This paper presents a reusable framework for configurable modules. Unlike most of the configurable modules found in the literature, the proposed framework is reusable over a wide range of applications. It provides a true reusable architecture at the RT level that is well suited for many soft IP implementations. The framework exploits a cellular structure that, due to its regularity, offers a good configurability over a range of applications. Among its main advantages, the framework incorporates a predefined and simple control functionality that reduces the design effort.

The framework was used with one of our applications called the Crozier frequency estimator, which is typical DSP algorithm. It is currently used in another project involving video encoding algorithms. Although it was only tested with very few applications, the framework is clearly applicable to board class of DSP algorithms.

This paper first presents the overall functionality of the proposed framework. The actual reusable architecture is then presented along with a practical application to the Crozier frequency estimation algorithm. Effectiveness with which a configurable Crozier module can be implemented is demonstrated.

2. The Framework Architecture

2.1 Basic functionality

The main property of a reusable framework is that it should be highly configurable while offering a generic architecture.

To achieve these characteristics, the framework employs a cellular architecture like the one depicted in figure 1. All the cells have the same structure that incorporates a controller (**C**), a memory element (**M**) and a processing operator (**O**). The basic function of a cell is to empty its memory element content into its operator. The operator performs mathematical operations on the data set. When processed, the data set is immediately sent to the memory element of the next cell. This last cell takes the data set to its own operators and propagates it to another cell. Assembling several cells together forms a pipelined datapath where the data set flows from memory element to memory element, executing while transferring, a set of mathematical functions or operations. Propagating the data set along the entire path implements the desired functionality distributed over several operators.

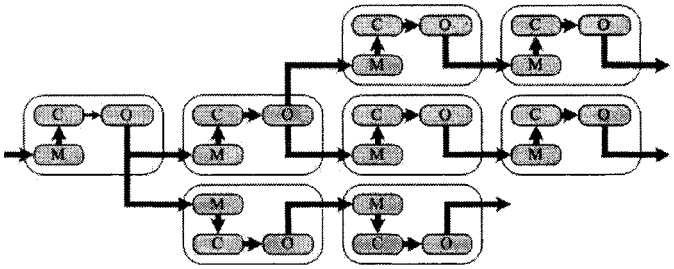

Figure 1 – General architecture of the reusable framework

The controller manages the data set transfers between memory and operator and between adjacent cells. Like in many designs, control is very crucial to overall functionality. Its implementation often represents a delicate part of the design. By using the memories empty status flags to synchronize both transfer operations, the complexity of control is considerately reduced. Actually, a unique controller can be used for all cells. Basically, if the destination memory is **empty**, the controller **empties** its own memory to its operator. Using memories properties to synchronize transfer operations makes the controller architecture unique and independent of the operators' content.

This very simple control strategy is a key feature of the framework. The control is predefined and independent of the considered application. This eliminates design of controllers and reduces design of configurable modules to describing the data operator

contents. Such operators are mostly conventional mathematical calculations that could be taken from a pre-implemented library.

Furthermore, utilizing a simple and unique communication protocol for all cells allows easy cell interconnection to form various configurations. This is how high configurability is achieved.

2.2 The Cell

Figure 2 shows the detailed architecture of the cell. It comprises a controller (**C**), a processing operator (**O**), two memory elements (**M1** and **M2**) and a set of multiplexers. As mentioned before, the general cell function is to transfer the content of one of its memory elements to its operator. The processed data set is immediately sent to other cells. The two memories are necessary to keep the pipeline busy: when a memory element is read, the other remains available to other incoming cells data sets. Such double buffering increases cost, but it decouples communicating cells. Multiplexers direct the dataflow in and out of memories.

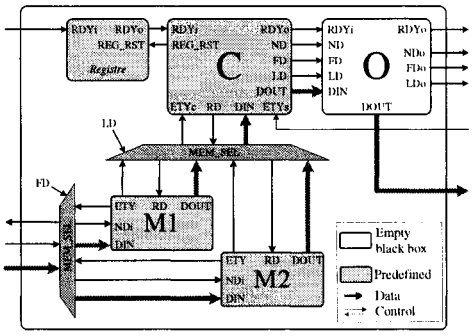

Figure 2 – Cellular architecture

The transfer procedure is implemented in a state machine that contains only a few states: the state diagram is depicted in figure 3. The coordination of the transfer procedure relies mainly on the empty status flag of the memory elements and the ReaDY (RDY) signal generated by the operators. The procedure starts when the cell detects the RDYi signal (ReaDY in) and it stops when its memory element is empty (ETYc, EmpTY current). The operator generates the RDYo (ReaDY out) flag when the data set

is completely processed. To perform a transfer, the destination memory element has to be free (ETYs, EmpTY subsequent). The state machine generates the FD (First Data), ND (New Data) and LD (Last Data) flags that are used to delimit the data set. These flags are used to synchronize the operator functions and control the multiplexers that manage the data flow. Notice that this property makes processing of the data set entirely independent from the number of samples. For many applications, and especially in signal processing, this is a significant advantage.

Another very interesting property of the controller is that it is absolutely free from counters or any sort of vector manipulation: adders, comparators, shifters, etc. Basically, the controller is a simple state machine coordinated with the memories properties (empty status signals). However, it is obvious that at some point, a form of counter is necessary to generate the empty signals. Putting it in the controller or in the memories is only a matter of system organisation. But it is this organization that facilitates design reuse; first, any common FIFO has an empty status signal; second, it is the controllers, and not the FIFOs, that complicate design. Also, this organization of functionalities provides complete independency between control and datapath. This means that all the technology dependant functionalities (adders, counters, multipliers and others) are regroup in the same component: the operator. Thus, transferring from a technology to another only implies localised modification of the design. This is a true advantage for reuse capabilities of the framework.

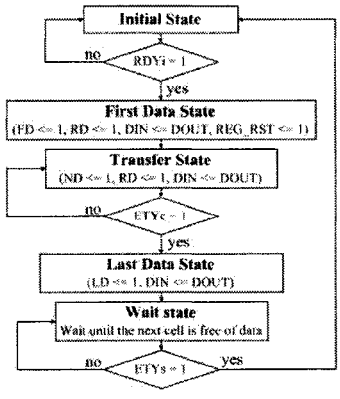

Figure 3 – State diagram of the controller

As illustrated in figure 2, most of the cell components are predefined (gray box). Only the operator content has to be designed (white box). To ensure that the operator will fit in the architecture, a wrapper is provided. The wrapper architecture comprises some delay lines and an empty black box (see figure 5). This defines the basic framework from which the hardware designer develops each cell. With the empty black box, the architecture remains generic for a variety of applications that match the supported data flow. The operator can be anything from a single mathematical operator to a more complex IP. The delay lines are used to transmit control signals through the operator and out of the cell. They synchronize data transfers and write operations to the next cell memory. They can also be used to manage the operators.

As for the memory elements, they can be anything from a simple register to a complete RAM or FIFO. The only characteristic required for the memories is that they generate the empty status signal. As discussed above, the FIFOs fit naturally in the cell architecture.

2.3 Iterative cell

Iterative functions or operations appear very often in signal processing applications. An iterative cell can easily be built by adding, to the original cell architecture, a third memory element and a set of multiplexers. The additional memory allows performing internal calculations through the iterations, while keeping the intercell communications active. For example, memories 1 and 2 can be used to iterate on the data set while memory 3 receives samples from the previous cells. The set of mux/demux are used to switch between the memory elements in order to route the data flow from the preceding cell, into the current cell and to the next cell. The controller of the iterative cell is easily obtained by adding states that will encapsulate the original transfer procedure: First Iteration State and Last Iteration State.

3. Case study

This section presents the implementation of the Crozier frequency estimator [6][7] with the proposed framework. The goal is to create an estimator that is configurable with

respect to some parameters. Mostly two reasons motivate the use of this application as an implementation example. First, the estimator has enough complexity, from both mathematical and control perspective, to demonstrate the framework efficiency. Second, previous work on the Crozier hardware [8] implementation allows evaluating the benefits of using the proposed framework.

3.1 The Crozier algorithm

Fast and precise frequency estimation of single-tone digitized signals is often crucial for applications in communication, radar, sonar and electronic warfare. Varieties of estimators have been developed over the years to satisfy numerous requirements. The Crozier algorithm is one among others that has the ability to yield accurate estimates from short signal segments. The Crozier estimator is particularly accurate over a wide range of frequencies [6][7][8].

Consider a segment of N complex signal samples (S_0, S_1,..., S_{N-1}). The core estimation is based on the delay-multiply-add technique (DMA) that calculates phase differences between successive samples.

$$\hat{Z} = \sum_{n=0}^{N-1-d} S_n^* S_{n+d} \tag{49}$$

where d is a user defined delay, expressed in sample periods. The result gives a unit-amplitude complex phasor representing the phase rotation over d sample periods of the sampling rate. Taking the angle of the d-th root of $\hat{z}$ yields the frequency estimate:

$$\omega = \angle\left(\hat{Z}\right)^{1/d} \tag{50}$$

Crozier devised a multi-branches algorithm. Figure 4 shows a block diagram of the Crozier algorithm. The algorithm uses the rotate-add-decimate (RAD) processing to decimate the signal between successive branches. This technique is used instead of regular decimation to further improve accuracy [7][8]. A sort of weighted averaging calculation is used to compile the successive Z estimates.

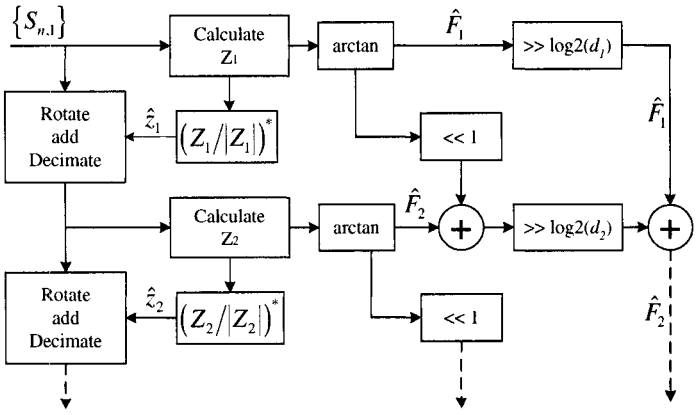

Figure 4 - Block diagram of the Crozier algorithm

The algorithm is composed of the 5 following functions: DMA, RAD, Normalization, Arctan and Averaging. For the *b*-th branch the equations are

$$Z_b = \sum_{n=0}^{N_b-1} S_{n,b}^* S_{n+1,b} \tag{51}$$

$$S_{n,b+1} = S_{2n-1,b} + \hat{z}_b^* S_{2n,b} \tag{52}$$

$$\hat{z}_b = Z_b / |Z_b| \tag{53}$$

$$\hat{P}_b = \angle Z_b \tag{54}$$

$$\hat{F}_b = \hat{F}_{b-1} + (\hat{P}_b + 2\hat{P}_{b-1})/d_b \tag{55}$$

where b=1,2,..., $B+1$, $d_b=2^{b-1}$ and $N_b=N/d_b$. The number of iterations is related to the number of samples with $B=\log_2(N/3)$.

3.2 Hardware Implementation

To implement the Crozier algorithm with the proposed framework, the first task is to construct the operators. In this case, the operators are obtained by implementing equations (3) to (7). As an example, we demonstrated the implementation of a generic DMARAD operator (see in figure 5) that implements the DMA and RAD functions.

The choice of this operator is not arbitrary: unlike the other operations, the DMA and RAD functions ((3) and (4)) are performed on all signal samples. Therefore, the processing time of these functions determines the performance of the estimator. A single DMARAD branch is obtained by concatenating functions RAD and DMA on a pipelined datapath. Despite a larger latency, the processing time is considerably reduced. Using parallel datapaths can further reduce processing time [8]. Since both DMA and RAD functions are made of complex operations, the resulting hardware structure will be an array of complex multipliers, complex adders and complex accumulators. To fit into the cell architecture, the only components to add are the delay lines. Synchronized with the datapath, they transmit the control signals (RDY, LD, ND and FD) to other cells. They can also be used to manage some operations. In this case, we use the New Data (ND) and Last Data (LD) flags to set and reset the accumulators. As mentioned above, this makes the DMARAD processing independent of the dataset width (N_b). It is a very important feature that will lead to good configurability.

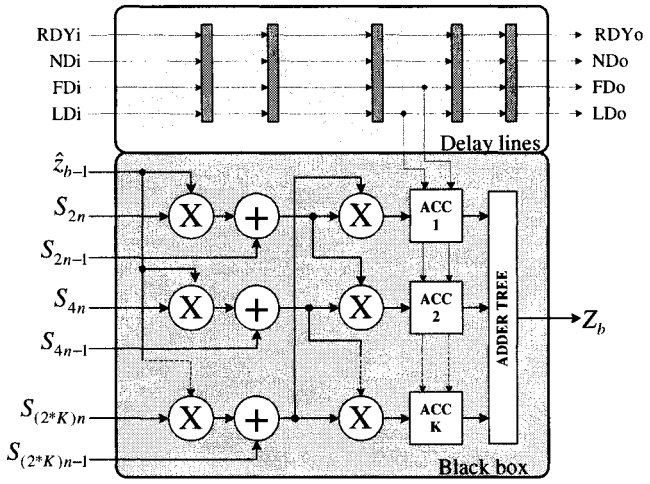

Figure 5 – DMARAD operator

The construction of the remaining operators is performed along the same method: implementation of the mathematical function and insertion of the delay lines. Note that for the Arctan function (6) we use the CORDIC algorithm [9]. Also, the Normalization operator (5) can be replaced with a scaling function that simply scales down the complex

components to approximately 1. The impact on the accuracy of the frequency estimation of an approximate scaling is negligible [8]. Usually, this function is included in DMARAD and DMA operators.

Once the various data operators are defined, the only task that remains is to assemble the cells into a configurable estimator. Figure 6 shows an interesting architecture of a configurable Crozier estimator. The architecture describes a cascade of stages that each implements a single branch of the algorithm and one other stage that implements an iterative Crozier. Each stage comprises DMARAD (or DMA), Averaging and Cordic cells. The specific number of single branch is implemented through a HDL generic parameter. The role of the iterative Crozier is to calculate the remaining iterations depending on the number of single branch stages generated and the data set width ($B=\log_2(N/3)$). The cascade of stages forms a pipeline where several data pulses are processed simultaneously. The term data pulse refers to a segment of N complex samples that corresponds to a significant event we wish to process with the algorithm. In many real time applications, the input signal takes the form of a flow of successive pulses that arrive at certain rate. Thus, it is necessary to have an estimator that delivers sufficient processing rate.

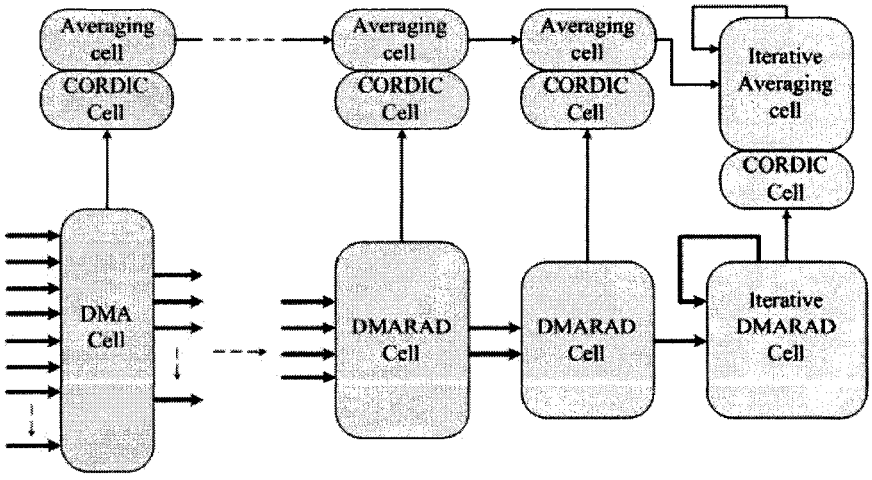

Figure 6 – Architecture of a configurable estimator

Notice that each stage contains a generic DMARAD operator. Since the number of samples decreases at each branch, the degree of parallelism implemented in the DMARAD cells decreases accordingly.

The architecture uses 4 operators: DMARAD, DMA, CORDIC and Averaging; as well as two cell formats: standard and iterative. The configurable estimator takes the form of a generic cells assembly pattern that can automatically generate different configurations. Thus, in a matter of minutes, a designer can increase/decrease the pulse processing rate and latency by generating an estimator with a different number of stages (with the iterative Crozier) and input parallelism data processing. Since all cells have the same interface and communication protocol, creating generic cells pattern is reduced to generating the appropriate cells and connections. This is a very easy task estimated to less than a day-person of work.

A primary hardware implementation of the Crozier algorithm was presented in [8]. The architecture was implemented in a single stage used iteratively. The basic stage was designed for a single data set length (96 samples) and the DAMRAD module employs a fixed parallelism of degree 3 in the datapaths. This implementation was derived without using the generic reusable framework. It used a single central controller that manages all of the activities. The controller area was 239 Luts (XCV1000 speedgrade –5). The estimated operating frequency for the controller was 80 MHz. With that implementation, the latency for processing a 96 samples pulse is 155 clock cycles. The rate at which it can process pulses is one per 155 clock cycles. A second design was developed using the generic framework. It is composed of 3 stages, which represents an equivalent hardware complexity. Unlike the previous version, this estimator can process any data set length. For a 96 samples pulse, the latency is 242 clock cycles and the minimum pulse repetition time is 96 cycles. The configuration uses 9 controllers for a total area consumption of 62 Luts. The estimated operating frequency for the controllers is 192 MHz (XCV1000 speedgrade –5).

It is obvious that with more stages, the pulse processing rate decreases while latency increases. But, the reduction in controllers complexity is remarkable when the generic implementation framework is used. It shows how the framework offers a simple and efficient control strategy that facilitates design reuse.

4. Conclusion

A reusable framework for configurable DSP modules has been presented. Advantages derived from using this architecture have been examined. It was shown that with predefined control issues, the implementation of a configurable module is reduced to the hardware description of the mathematical functions of the considered DSP application. It was also shown that with its cellular architecture, the framework facilitates development of configurable implementations. The framework was used to implement a frequency estimator based on the Crozier algorithm. The framework proved to be a very efficient method to design a configurable Crozier estimator. A generic implementation with configurable degree of parallelism and pipelining was easily and quickly assembled.

5. Acknowledgements

This work was supported financially by the Defence Research Establishment in Ottawa and a Canada Research Chair to one of the authors.

6. References

[1] M. Keating, P. Bricaud, *"Reuse Methodology Manual For System-On-A-Chip"*, Kluwer Academic Plublisher, 2000.

[2] Xilinx, "Xilinx Design Reuse Methodology for ASIC and FPGA Designers", www.xilinx.com.

[3] J.M. Chang, S.K. Agun, *"Designing reusable components in VHDL"*, Proceedings of the 13th Annual IEEE international ASIC/SOC conference, September 2000, pp165-169.

[4] Fanucci, L.; Saponara, S.; Cenciotti, A., *"IP reuse VLSI architecture for low complexity fast motion estimation in multimedia applications"*, Proceedings of the 26th Euromicro Conference, 2000.,Volume: 1 , 5-7 Sept. 2000, Pages:417 - 424 vol.1

[5] Sadasivam, M.; Sangjin Hong, *"Application specific coarse-grained FPGA for processing element in real-time parallel particle filters"*, The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings., June 30 - 2 July 2003, Pages:116 – 119.

[6] S.N. Crozier, *"Performance and Complexity of Discrete-Time Frequency Estimation Algorithms"*, 17th Biennial Symposium on Communications, Queen's University, Kingston, Ontario, Canada, May 30 – June 1, 1994.

[7] R. Mantha, S.N. Crozier, *" Implementation of a Delay-Multiply-Average frequency Estimator with Rotate-Add-Decimate Processing"*, 18th Biennial Symposium on Communications, Quenn's University, Kingston, Ontario, Canada, June 2 – June 5, 1996.

[8] L-P. Lafrance, M-A. Cantin, Y. Savaria, S.H. Sung, P. Lavoie, *"Architecture and performance characterization of hardware and software implementations of the Crozier frequency estimation algorithm*", The IEEE International Symposium on Circuits and Systems (ISCAS 2002), Phoenix, USA, May 2002, Vol. IV pp823-826.

[9] R. Andraka, *"A survey of CORDIC algorithms for FPGA based computers"*, Andraka Consulting Group, Inc, North Kingstown, Ontario,1998, 11 pp.